Joakim Koskinen

# WEB-KÄYTTÖLIITTYMÄ RENKAAN TARKASTUSASEMALLE

Tieto- ja viestintätekniikan koulutusohjelma

2019

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

WEB-KÄYTTÖLIITTYMÄ RENKAAN TARKASTUSASEMALLE

Koskinen, Joakim
Satakunnan ammattikorkeakoulu
Tieto- ja viestintätekniikan koulutusohjelma
Toukokuu 2019
Sivumäärä: 45

Asiasanat: reactjs, web-käyttöliittymä, tila

---

Tässä opinnäytetyössä luotiin Cimcorp Oy:lle web-käyttöliittymä renkaan tarkastusasemalle. Tarkastusasemassa on tarkoitus kirjata ylös renkaan laatu ja mahdolliset virheet. Työn keskeisimpiä asioita olivat viivakoodinlukija ja tilan hallinta. Käyttöliittymän toimintoja voitiin käyttää viivakoodinlukijalla. Tila määrittää miten komponentti renderöi ja käyttäytyy. Tilojen avulla voitiin luoda dynaamisia ja interaktiivisia komponentteja. Tilan hallinnassa käytettiin apuna react-redux kirjastoa/pakettia.

Käyttöliittymä tehtiin ja suunniteltiin Cimcorp Oy:n tuotekehityksen ja WebUI-tiimin kanssa. Toteutuksessa käytettiin muun muassa React-kirjastoa, Bootstrap-viitekehystä, Reduxsia ja Redux-saagaa. Versionhallintaohjelmana toimi Git, jonka apuna käytettiin Bitbucket-sivustoa. Aktiivinen versionhallinta helpotti käyttöliittymän siirtämistä Cimcorp Oy:n tuotekehitykseen.

Työn lopputuloksena luotiin React komponenttipohjainen web-käyttöliittymä ja viivakoodinlukijan käsittelyyn KeyInputListener-luokka. KeyInputListener-luokan tarkoituksena oli käsitellä skannattu viivakoodi etu- ja jälkiliitteen avulla. Erillinen luokka mahdollistaa viivakoodilukijan helpon käyttöönoton myös muissa web-käyttöliittymissä.

WEB INTERFACE FOR TIRE INSPECTION STATION

Koskinen, Joakim
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Information and Communication Technology
May 2019
Number of pages: 45

Keywords: reactjs, user interface, state

---

In this thesis web interface for tire inspection station was created for Cimcorp Oy. The purpose of the inspection station is to record the quality and possible errors of the tire. The most important things in this work were barcode scanner and state management. The user interface functions could be used with a barcode scanner. State determines how component renders and behaves. State were used to create dynamic and interactive components. React-redux library was used to help manage the state.

The user interface was made and designed with Cimcorp´s product development and WebUI team. For example, React library, Bootstrap framework, Redux and Redux-saga were used to implement the user interface. Git was used as version-control program with help of the Bitbucket site. Active version-control made it easy to transfer the user interface to Cimcorp´s product development.

The result of this study was a React component-based web interface and a KeyInput-Listener class for handling barcode scanner. The purpose of the KeyInputListener class was to handle the scanned barcode with the prefix and suffix. A separate class enables for easy deployment of barcode scanner in other web interfaces.

SISÄLLYS

# 1 JOHDANTO

Tämän työn tarkoituksena oli tehdä web-käyttöliittymä renkaan tarkastusasemalle. Työ tehtiin Cimcorp Oy:n tarpeista liikkeelle lähtien. Työhön kuului vain front-end puoli eli koodia ei tehty lainkaan back-end puolelle. Cimcorp Oy:n vanhassa/nykyisessä järjestelmässä löytyy tarkastusasemalle käyttöliittymä. Tässä työssä otettiin mallia tästä vanhasta käyttöliittymästä ja valmiista web-käyttöliittymän luonnoksesta. Työssä käytettiin React-kirjastoa, Bootstrap-viitekehystä, Reduxia ja Redux-saagaa. Versiohallintaohjelmana toimi Git, jonka apuna käytettiin Bitbucket-sivustoa. Uudessa web-käyttöliittymässä piti toteuttaa samat toiminnot, mitkä löytyivät vanhastakin käyttöliittymästä.

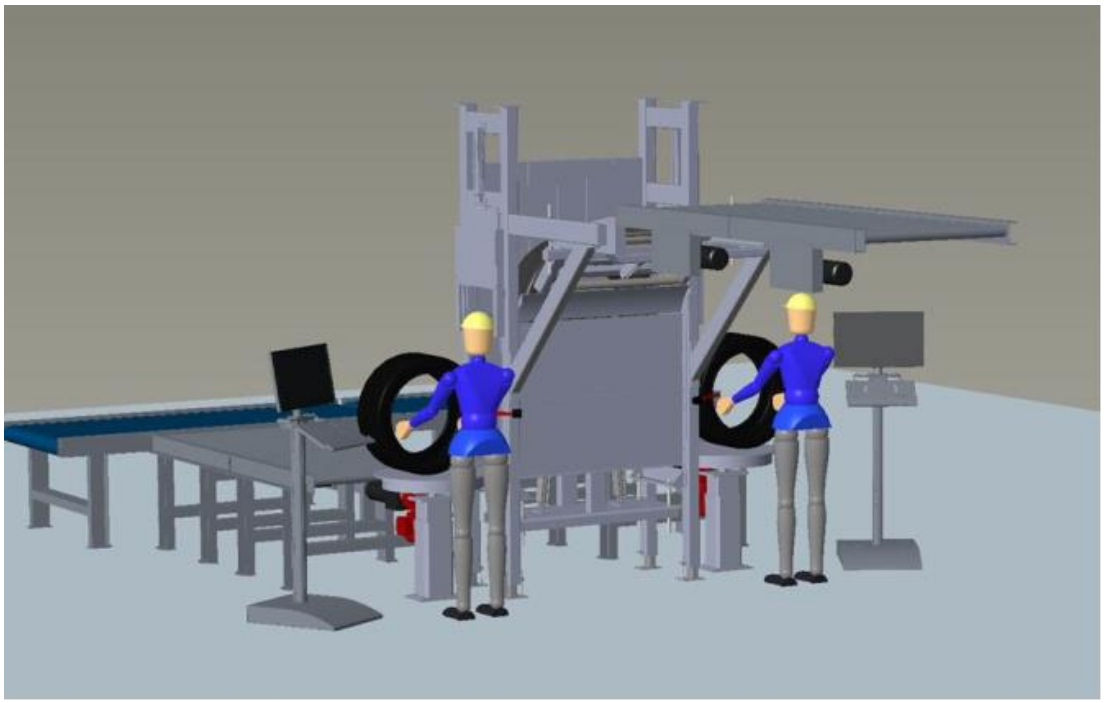
Työn tavoitteena oli toteuttaa valittujen tekniikoiden ja ohjelmointikielien avulla komponenttipohjainen web-käyttöliittymä. Käyttöliittymän keskiössä oli viivakoodinlukija ja järjestelmän tilojen hallinta. Melkein kaikki toiminnot haluttiin pystyä tekemään pelkällä viivakoodinlukijalla, joten tavoitteena oli melkein painallus riippumaton käyttöliittymä. Komponenttipohjaisuudella oli tarkoitus vähentää koodia ja mahdollisesti auttaa käyttöliittymän tulevaisuuden kehitystä.

Luvussa 2 selitetään, mikä on tarkastusasema, kuka työskentelee asemassa ja mitä hyviä puolia tarkastusasemassa on. Luvussa 2 kerrotaan myös tarkastusaseman toiminnallisuus. Luvussa 3 käydään lävitse käyttöliittymän front-end puoli ja kerrotaan mitä tekniikoita ja ohjelmointikieliä käytetään. Luvussa 4 käydään läpi työssä käytetyt versionhallintaohjelmistot. Luvussa 5 kerrotaan käyttöliittymän varsinainen toteutus, johon kuuluu testaus, asennus, tärkeiden komponenttien luonti, komponenttien tyyli, viivakoodin luku ja tilan hallinta. Luvussa 6 on lopuksi yhteenveto, jossa käydään lävitse työn tärkeitä asioita, ja mitkä olivat työn haasteita.

# 2 TARKASTUSASEMA

Rengastehtaassa tarkastusasema (Kuva 1) on asema, jossa tarkastetaan renkaan laatu. Tarkastusasemassa työskentelee operaattori, jonka tehtäviin kuuluu renkaan laadun tarkastus ja mahdollisten laatuvirheiden ja laatuluokan kirjaaminen tietokantaan. Laatuvirheet kirjataan mahdollisimman tarkasti tarkastusaseman käyttöliittymän valintoja käyttäen. Tarkastusasemasta löytyy yleensä näyttö, näppäimistö, hiiri ja viivakoodinlukija. Näyttö voi olla joko normaali näyttö tai kosketusnäyttö. Riippuen projektista tarkastusaseman ulkonäkö, laitteet ja laitteiden määrä voivat vaihdella. Useimmiten tarkastusasema sijaitsee rengastehtaan loppuvaiheilla. Tarkastusasema on tärkeä osa rengastehdasta, koska siinä nähdään koko tuotannon laatu.

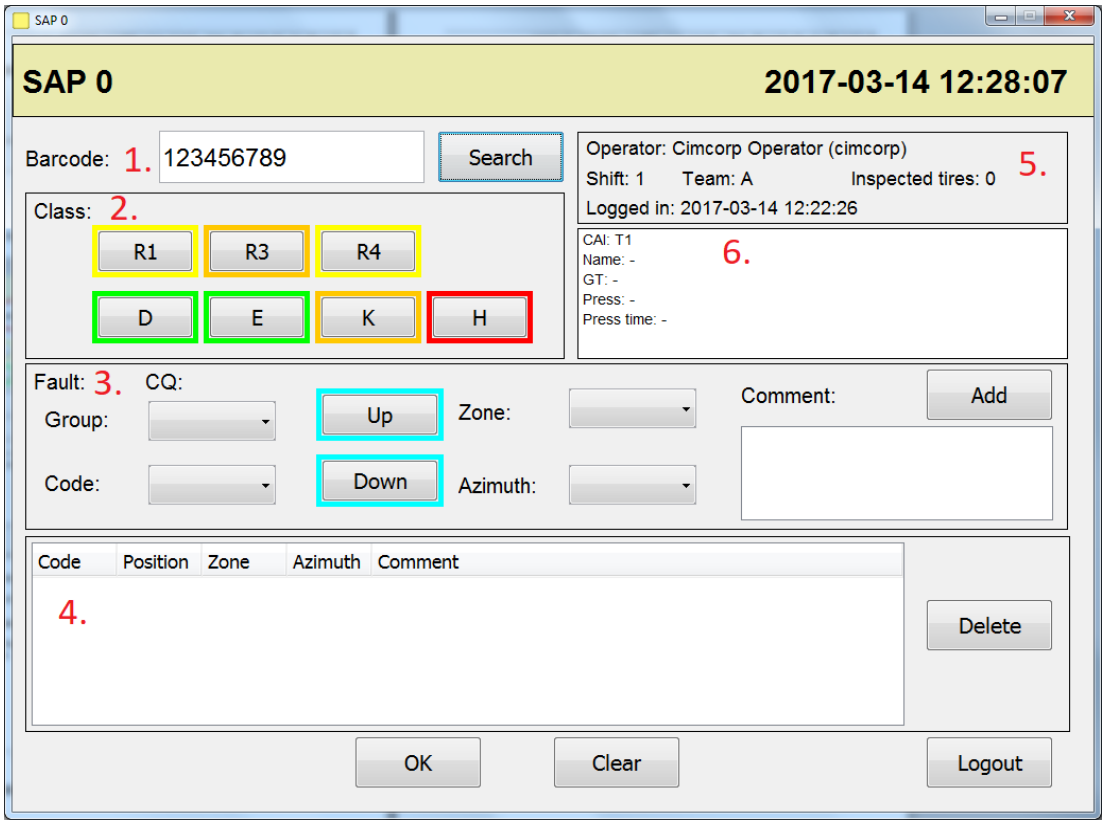
Tarkastusaseman yksi tärkeimmistä hyödyistä on datan kerääminen. Tarkastusasemasta saadun datan avulla voidaan esimerkiksi lokeroida kaikki renkaat laatuluokan mukaan. Vialliset renkaat voidaan joko poistaa tuotannosta tai korjata. Datasta voidaan myös tehdä erilaisia analyyseja, joiden avulla voidaan parantaa laatua rengastehtaan tuotannossa. Esimerkiksi, jos rengastehtaan tuotannosta tulee paljon huonolaatuisia renkaita, se huomataan viimeistään tarkastusasemasta saadusta datasta.



Kuva 1. 3D-mallinnus tarkastusasemasta (Cimcorp Oy 2018)

## 2.1 Toiminnallisuus

Renkaan tullessa tarkastusasemalle rengastehtaan työntekijä eli operaattori skannaa tai näppäilee renkaassa olevan viivakoodin käyttöliittymälle ja viivakoodin syötettyä painaa Search-nappia (1.). Tämän jälkeen operaattori valitsee käyttöliittymästä renkaan laatua vastaavan laatuluokan (2.). Renkaasta löytyvät laatuvirheet operaattori kirjaa ja kommentoi mahdollisimman tarkasti, ja lisää laatuvirheet painamalla Add-nappia (3.). Lisätyt laatuvirheet kommentteineen näkyvät käyttöliittymän alhaalla olevassa taulukossa. Laatuvirheitä voi kirjata maksimissaan kolme ja esimerkiksi kirjoitusvirheen tullessa jo kirjatut laatuvirheet voidaan poistaa valitsemalla kyseinen laatuvirhe ja tämän jälkeen painamalla Delete-nappia (4.). Kohdassa 5. näkyy kyseiseen käyttöliittymään kirjautunut operaattori, työvuoro, tiimi, kuinka monta rengasta on tarkastanut ja milloin on kirjautunut sisään. Jos rengas löytyy tietokannasta, renkaan tiedot näkyvät kohdassa 6. OK-napin painalluksella kaikki kirjatut laatuvirheet tallentuvat tietokantaan. Clear-nappia painamalla kaikki tiedot käyttöliittymässä resetoidaan. Ja lopuksi Logout-nappia painamalla operaattori kirjautuu ulos käyttöliittymästä. (Kuva 2)




Kuva 2. Renkaan tarkastusaseman vanha käyttöliittymä (Cimcorp Oy 2018.)

# 3 FRONT-END

## 3.1 JavaScript

JavaScript on järjestelmäriippumaton olioperustainen skriptauskieli, jota käytetään interaktiivisten verkkosivujen tekemiseen. (Mozilla 2019.) Kun JavaScript luotiin sen alkuperäinen nimi oli LiveScript, mutta Java ohjelmointikieli oli tuolloin suosittu, joten LiveScript muutettiin JavaScriptiin. JavaScript ohjelmia kutsutaan skripteiksi. Nämä skriptit voidaan kirjoittaa suoraan HTML-koodiin ja suorittaa automaattisesti, kun verkkosivu latautuu. Skriptit toimitetaan ja suoritetaan tavallisena tekstinä. Kun JavaScript kehittyi siitä tuli täysin itsenäinen ohjelmointikieli, jolla on oma määrittely ECMAScript, ja nyt sillä ei ole mitään yhteyttä Java ohjelmointikieleen. (JavaScript.info 2019.)

## 3.2 ECMAScript

ECMA (The European Computer Manufacturer´s Association) on omaksunut ECMAScriptin skriptauskielien standardiksi. Laajasti käytettyjä skriptauskieliä, kuten JavaScript, Jscript ja ActionScript, kehitetään ECMAScript standardin perusteella. ECMAScript standardia kehitetään jatkuvasti, esimerkiksi nopeuttamaan verkkosovellusten kehitystä ja täyttämään verkkosovelluksien uudet kehityssuunnat. (Beattie 2018.)

ES6 eli ECMAScript 6 on kuudes versio skriptauskielen standardista. ECMAScript 6:n syntaksisääntöjen takia kehittäjien on helpompi ohjelmoida monimutkaisia verkkosovelluksia hyödyntämällä uusia tietotyyppejä, metodeja, avainsanoja, moduuleja ja luokkia. ECMAScript 6 samalla myös nopeuttaa koodausta antamalla niin sanottuja ”oikoteitä”, esimerkiksi lyhentämällä jonkin jo olemassa olevan asian syntaksia. (Beattie 2018.)

Käytetyimmät ECMAScript 6:n ominaisuudet ovat let ja const, spread-operaattori ja nuolifunktiot. Let ja const ovat uusia avainsanoja määrittelemään muuttujat. Kun näitä käytetään määrittelemään muuttujia, niiden niin sanottu näkyvyysalue on rajattu

lohkoon eikä esimerkiksi funktioon. Tämä tarkoittaa sitä, että nämä muuttujat ovat käytettävissä vain siinä lohkossa, jossa ne ovat määritelty. Let muuttujien tietoa voidaan muuttaa uudelleen, mutta let muuttujia ei pysty määrittelemään uudelleen samassa näkyvyysalueessa. Const muuttujiin täytyy määritellä niin sanottu vakioarvo, ja const muuttujia ei myöskään pysty määrittelemään uudelleen samassa näkyvyysalueessa. ECMAScript 6:n spread-operaattori merkitään kolmella pisteellä (…), spread-operaattoria käytetään laajentamaan iteroituvia objekteja useisiin elementteihin (Kuva 3). ECMAScript 6:n nuolifunktiot käyttäytyvät hyvin samankaltaisesti kuin normaalit funktiot, mutta ne eroavat syntaktisesti toisistaan (Kuva 4). (Atto 2018.)

```
const cities = ["Kampala", "Nairobi", "Lagos"];
console.log(...cities);

//output
Kampala Nairobi Lagos
```

Kuva 3. Esimerkki spread-operaattorista (Atto 2018.)

```
let HelloWorld = () => { console.log('Hello World') }
```

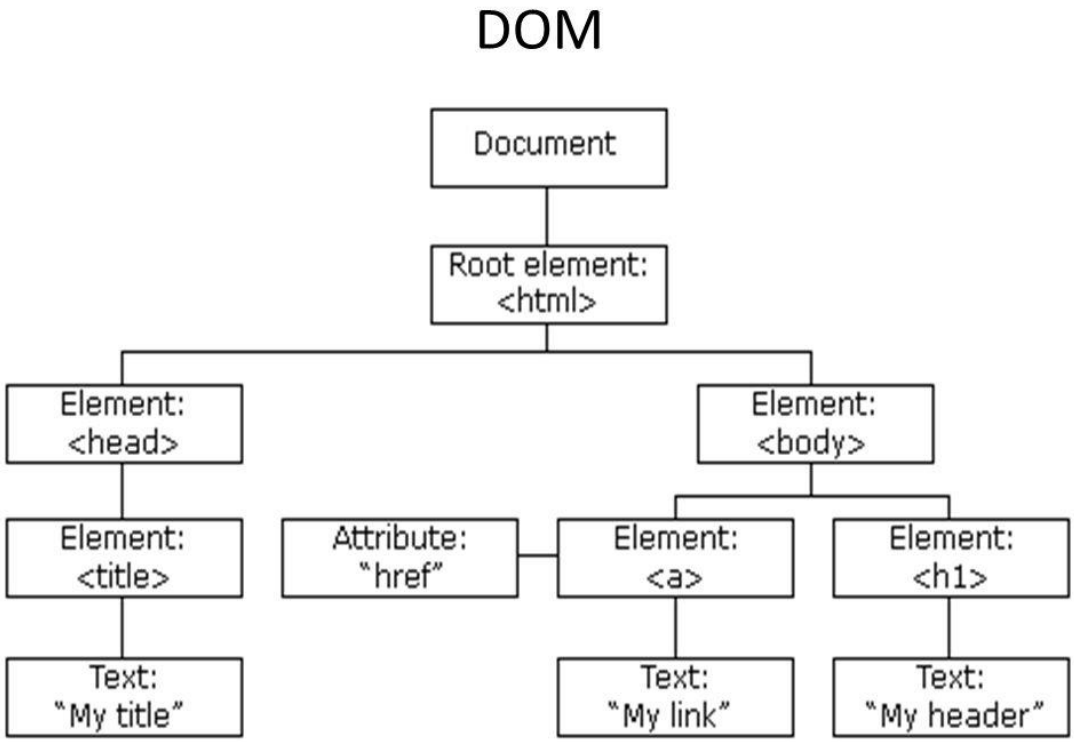
Kuva 4. Esimerkki nuolifunktion käytöstä (Shavin 2018.)

### 3.3 ReactJS

React on yksi suosituimmista JavaScript-kirjastoista, jota käytetään luomaan erilaisia käyttöliittymiä. Reactin on kehittänyt Facebook vuonna 2011, ja se on kehitetty käsittelemään laajoja ja data käyttöisiä verkkosovelluksia. Reactin keskeisempiä ominaisuuksia ovat komponenttipohjaisuus, virtuaalinen DOM, deklaratiivisuus ja JSX. (Banks & Porcello 2017, 1-2.) React on hyvin yksinkertainen ja kevyt JavaScript-kirjasto, joka käsittelee MVC-arkkitehtuurista vain ”View” osan. Reactin yksinkertaisuuden takia kuka tahansa JavaScript ohjelmistokehittäjä voi ymmärtää perusteet ja aloittaa monipuolisten verkkosovelluksien kehittämisen vain muutaman opiskelupäivän jälkeen. (Sahin 2017.)

### 3.3.1 Virtuaalinen DOM

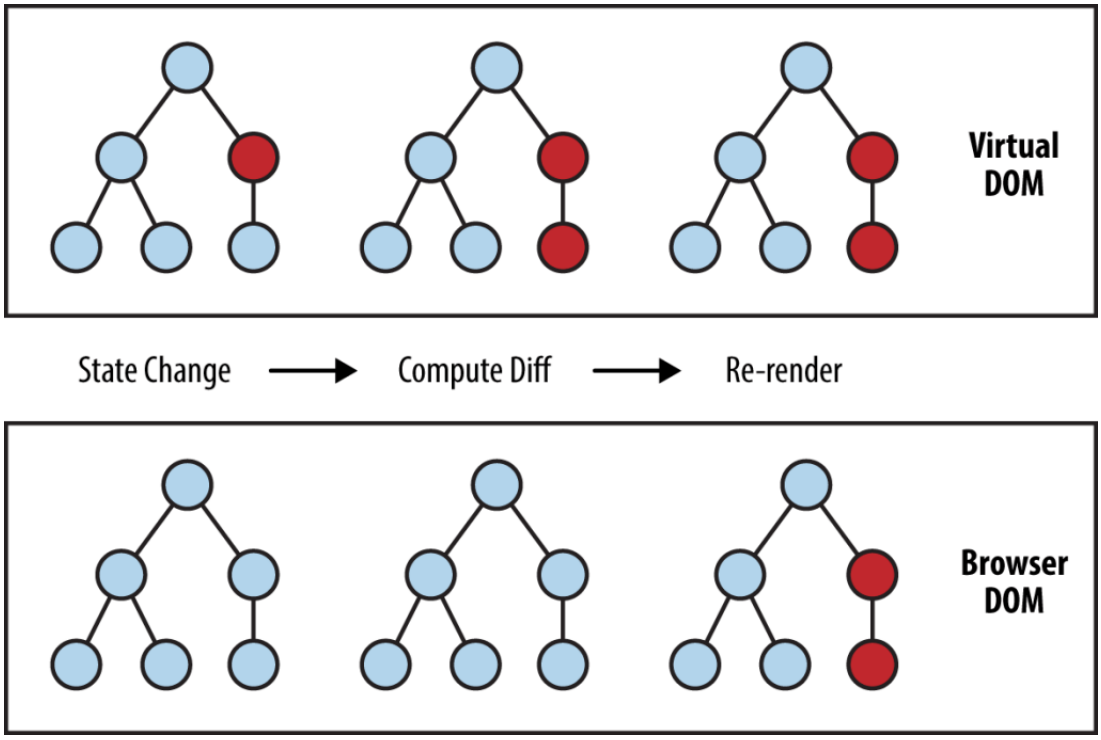
DOM eli Document Object Model edustaa verkkosovelluksen käyttöliittymää. Joka kerta kun verkkosovelluksen käyttöliittymän tilassa tapahtuu muutos, DOM päivittyy vastaamaan kyseistä muutosta. Haittapuolena on se, että kun DOM:ia käsitellään useasti, se vaikuttaa DOM:in suorituskykyyn, jolloin se hidastuu. DOM kuvataan puumaisena tietorakenteena (Kuva 5), tämän takia DOM:iin tehdyt muutokset ja päivitykset ovat nopeita. Muutoksen jälkeen päivitetty elementti ja sen lapset täytyy renderöidä uudelleen, jotta käyttöliittymä päivittyy. Käyttöliittymän uudelleen renderöinti tekee DOM:in manipuloinnista hidasta. (Ravichandran 2018.)




Kuva 5. DOM-malli puu tietorakenteessa (Rich 2015.)

Virtuaalinen DOM toimii huomattavasti paremmin kuin ”oikea” DOM. Aina kun verkkosovelluksen tila muuttuu, virtuaalinen DOM päivitetään todellisen DOM:in sijaan. Virtuaalinen DOM luodaan aina, kun käyttöliittymään lisätään uusia elementtejä. Virtuaalinen DOM esitetään puumaisena tietorakenteena, jossa jokainen elementti on solmu. Jos näiden elementtien tila muuttuu, luodaan uusi virtuaalinen DOM-puu. Sitten tätä puuta verrataan edelliseen virtuaaliseen DOM-puuhun. Kun

tämä vertailu on tehty, virtuaalinen DOM laskee parhaan mahdollisen metodin näiden muutosten tekemiseksi todelliseen DOM:iin. Näin varmistetaan, että todelliseen DOM:iin tehdään mahdollisimman vähän toimintoja, jolloin suorituskyky paranee. Kuvassa 6 nähdään virtuaalinen DOM-puu ja vertailurosessi. Punaiset ympyrät edustavat solmuja, joiden tila on muuttunut. Tämän jälkeen lasketaan edellisen virtuaalisen DOM-puun ja nykyisen virtuaalisen DOM-puun välinen ero. Laskennan jälkeen koko parent alipuu renderöidään uudelleen, jotta saadaan päivitetty käyttöliittymä. Tämä uusi puu päivitetään sitten todelliseen DOM:iin. (Ravichandran 2018.)




Kuva 6. Virtuaalisen DOM-puun vertailuprosessi (Eisenman 2015)

### 3.3.2 Komponentti

React tarjoaa komponenttipohjaisen rakenteen käyttöliittymille. Komponentit ovat kuin legopalikoita. Yksinkertaisesti sanottua komponentti on JavaScript luokka tai funktio, joka valinnaisesti hyväksyy syöttöjä eli ominaisuuksia(props) ja palauttaa React-elementin, joka kuvaa kuinka käyttöliittymän osan pitäisi näkyä. Komponentit ovat minkä tahansa React-sovelluksen rakennuspalikoita, tyypillinen React-sovellus sisältää monia komponentteja. (Kagga 2018.)

Funktionaaliset komponentit ovat yksinkertaisesti JavaScript funktioita, jotka voivat saada tietoja parametreina. (Kagga 2018.) Kuvassa 7 nähdään esimerkki funktionaalisesta komponentista, joka on kirjoitettu käyttäen ES6:n nuolifunktiota.

```
const Greeting = () => <h1>Hi, I'm a dumb component!</h1>;
```

Kuva 7. Esimerkki funktionaalisesta komponentista (Kagga 2018.)

Luokkakomponentit luodaan käyttämällä ES6:n luokka syntaksia. Verrattuna funktionaaliseen komponenttiin luokkakomponentilla on joitakin lisäominaisuuksia kuten kyky sisältää logiikkaa ja paikallinen tila. Luokkakomponentin sisällä voivat esimerkiksi olla metodit, jotka käsittelevät onClick-tapahtumia. Paikalliseen tilaan voi esimerkiksi tallentaa erilaisia arvoja. (Kuva 8)

```
class Greeting extends React.Component {
  render(){
    return <h1>Hi, I'm a smart component!</h1>;
  }
}
```

Kuva 8. Esimerkki luokkakomponentista (Kagga 2018.)

### 3.3.3 Deklaratiivinen ohjelmointi

Deklaratiivinen ohjelmointi on kuin maalauksen kuvaileminen, kun taas imperatiivinen ohjelmointi on ohjeet tämän maalauksen tekemiseen. Deklaratiivinen ohjelmointi tekee koodista luettavampaa, ohjelmaa on helpompi lukea, koska se piilottaa alemman tason yksityiskohdat. Deklaratiivisessa ohjelmoinnissa ei edes tiedetä järjestelmän alemman tason yksityiskohdista. Deklaratiivinen ohjelmointi tekee koodista myös helpommin ymmärrettävää, koska se on hyvin tiivis ja kuvaa ratkaisua menettelyn sijasta. Deklaratiivisia ohjelmointikieliä ovat muun muassa SQL, CSS ja React. (Sharma 2018.)

Reactin avulla tehdään interaktiivisia käyttöliittymiä muuttamalla komponenttien tilaa, ja React huolehtii DOM:in päivittämisestä tämän muutoksen mukaan. Kuvassa 9

nähdään koodi, joka luo tekstin tilana, ja tämän tekstin voi muuttaa napin painalluksena ”Hello Worldiksi”.

```
import React from "react";

class App extends React.Component {
  state = {
    message: "Hello react"
  }
  render() {
    return (
    <div className="App">
      <h1>{this.state.message}</h1>
      <button
        onClick={e => this.setState({message: "Hello World"})}>
          Change Message</button>
    </div>
  );
  }
}
```

Kuva 9. Esimerkki tilallisesta komponentista (Sharma 2018.)

Reactissa DOM on deklaratiivinen. Tämä tarkoittaa, että DOM:iin ei olla koskaan vuorovaikutuksessa. Käyttöliittymä päivitetään, kun tila muuttuu, mikä helpottaa käyttöliittymän suunnittelua ja virheenkorjausta. Tilan voi muuttaa, ja katsoa kuinka käyttöliittymä muuttuu sen mukana. (Sharma 2018.)

### 3.3.4 JSX

JSX on JavaScript laajennus, jonka avulla voidaan määritellä React-elementtejä. JSX syntaksi näyttää hyvin samanlaiselta kuin HTML ohjelmointikielen syntaksi. Facebook React-tiimi julkaisi JSX:n samaan aikaan, kun he julkaisivat Reactin. Facebook halusi toimittaa lyhyemmän syntaksin monimutkaisten DOM-puiden luomiseen attribuuteilla, he toivoivat myös tekevänsä Reactista paremmin luettavampaa kuten HTML ja XML. (Banks & Porcello 2017, 81-83.)

JSX:ssä React-elementin tyyppi on määritelty tunnisteen(tag) avulla. Tunnisteiden attribuutit edustavat ominaisuuksia. Elementin ”lapset” voidaan lisätä aloitus ja lopetus tunnisteiden väliin. (Banks & Porcello 2017, 81-83.) (Kuva 10)

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

Kuva 10. Esimerkki JSX-syntaksista ja mihin muotoon se kääntyy (ReactJS 2019.)

### 3.3.5 Babel

Useimmat ohjelmistokielet mahdollistavat lähdekoodin kääntämisen. JavaScript on tulkittavaa kieltä: selain tulkitsee koodin tekstiksi, joten JavaScriptiä ei tarvitse erikseen kääntää. Kaikki selaimet ei kuitenkaan tue uusimpia ECMAScript versioita, kuten ES6 ja ES7, ja mikään selain ei tue JSX-syntaksia. Tämän takia tarvitsemme keinon kääntää JavaScript koodin sellaiseksi, että selain ymmärtää sen. Tätä prosessia kutsutaan transpilingiksi ja se on, mitä Babel on suunniteltu tekemään. Babel tulee aina virallisen ”create-react-app” komennon/paketin mukana, joten sitä ei tarvitse erikseen ladata. (Banks & Porcello 2017, 84.) Kuvassa 11 nähdään kuinka Babel muuttaa ES6 käytetyn nuolifunktio-syntaksin takaisin vanhempaan versioon.

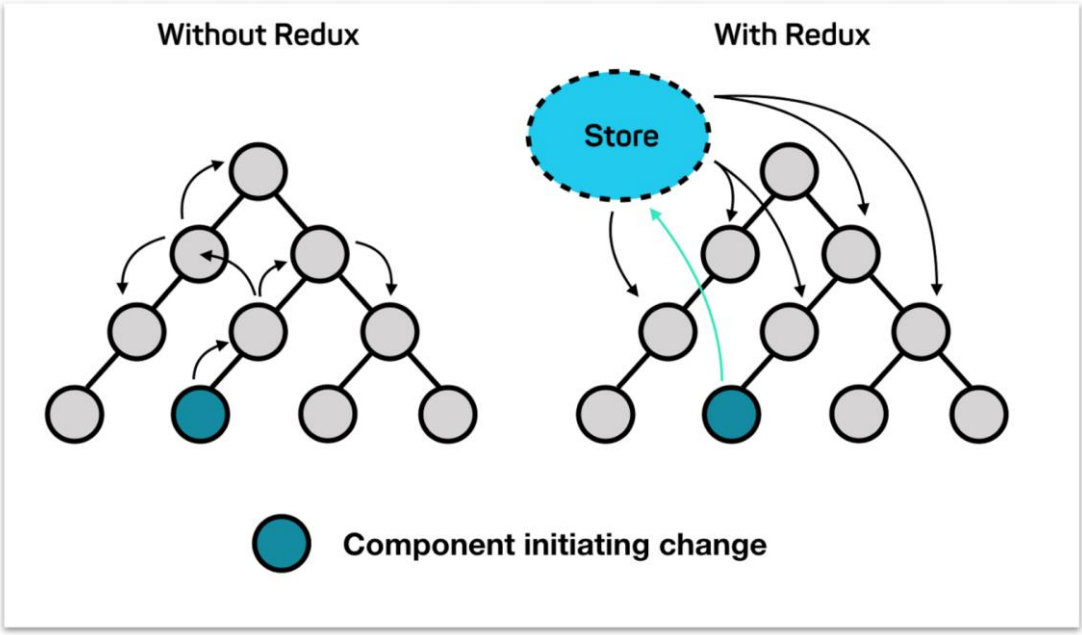
```
JavaScript

// Babel Input: ES2015 arrow function
[1, 2, 3].map((n) => n + 1);

// Babel Output: ES5 equivalent
[1, 2, 3].map(function(n) {
  return n + 1;
});
```

Kuva 11. ES6 nuolifunktio-syntaksin muunnos (BabelJS 2019.)

## 3.4 ReduxJS

Redux on ennustettavissa oleva tilasäiliö (state container) JavaScript-sovelluksille (Kuva 12). Redux auttaa kirjoittamaan sovelluksia, jotka käyttäytyvät johdonmukai-

sesti, toimivat eri ympäristöissä ja joita on helppo testata. (ReduxJS 2018.) Redux on hyvä ottaa käyttöön, kun sovelluksen koko kasvaa suureksi. Reduxin käyttö pienimmissäkin sovelluksissa on suotavaa, koska se auttaa hallitsemaan tiloja. Yksinkertaisesti sanottuna Redux on tilojen hallintatyökalu. Vaikka Reduxia käytetään enimmäkseen Reactin kanssa, sitä voidaan käyttää myös monen muun JavaScript-kehyksen tai kirjaston kanssa. Reduxin avulla sovelluksen koko tila säilytetään ”storessa” ja jokainen komponentti voi käyttää mitä tahansa tilaa, jota se tarvitsee tästä ”storesta”. (Ighodaro 2018.)




Kuva 12. Redux-säiliö (Weck 2017.)

Redux ei tule esimerkiksi ”create-react-app” paketin mukana, vaan se täytyy erikseen ladata ja asentaa. (Kuva 13)

```
npm install --save react-redux
```

Kuva 13. Npm-komento react-redux paketin lataamiseen ja asentamiseen (ReduxJS 2019.)

Kuvassa 14 nähdään react-reduxin datan kulku. Store varastoi kaikki sovelluksen tilat JavaScript-objektiksi. Provider tekee ”storen” saatavilla olevaksi. Container hankkii sovelluksen tilan ja toimittaa sen ominaisuutena(prop) komponenteille. Käyttäjä on vuorovaikutuksessa näkymä komponentin kautta. Actionit aiheuttavat muu-

toksia ”storessa”, ne voivat muuttaa sovelluksen tilaa. Reducer, ainoa tapa muuttaa sovelluksen tila, hyväksyä tila ja toiminta(action) sekä palauttaa päivitetty tila. (Karwal 2018.)


Kuva 14. React-redux data-flow (Tutorialspoint 2019.)

## 3.5 Redux-saga

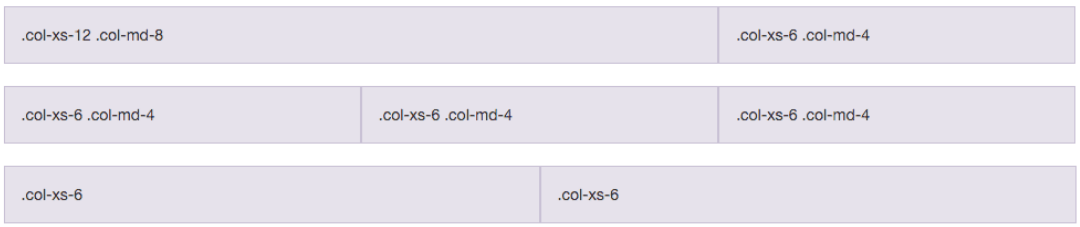
Redux-saga on kirjasto, jonka tarkoituksena on helpottaa sovellusten sivuvaikutuksien hallintaa, testattavuutta, virheiden käsittelyä ja tehostaa suoritusta. Ajatusmalli on se, että redux-saga on kuin erillinen säie sovelluksessa, joka on yksin vastuussa sivuvaikutuksista. Esimerkkejä sovelluksien sivuvaikutuksista ovat muun muassa asynkroniset asiat, kuten datan hakeminen, ja epäpuhtaat asiat, kuten selaimen välimuistiin pääseminen. Redux-saga on Reduxin väliohjelmisto, joka tarkoittaa sitä, että tätä säiettä voidaan käynnistää, keskeyttää ja peruuttaa pääsovelluksesta normaaleilla Redux actioneilla, sillä on myös pääsy Redux sovelluksen tilaan ja se voi myös lähettää(dispatch) Redux actioneja. (Redux-saga 2019.)

## 3.6 CSS

CSS eli Cascading Style Sheetsin avulla kuvataan verkkosivujen esitystapaa, esimerkiksi muuttamalla värejä, ulkoasua ja fontteja. CSS on riippumaton HTML:stä ja sitä voidaan käyttää minkä tahansa XML-pohjaisen merkintäkielen kanssa. HTML:n erottaminen CSS:stä helpottaa verkkosivustojen ylläpitoa, tyyliohjeiden jakamista sivujen välillä ja sivujen mukauttamista eri ympäristöihin. (Mozilla 2019.)

## 3.7 Bootstrap

Bootstrap on yksi suosituimmista HTML, CSS ja JavaScript viitekehyksistä, jolla kehitetään responsiivisiä projekteja verkossa. Ruudukko (Grid) on luultavasti yksi viitekehyksen olennaisimmista näkökohdista (Kuva 15). Se on perusta, johon koko ulkoasu luodaan. Tämän lisäksi Bootstrapin ydin CSS:n avulla voidaan lisätä erilaisia hyödyllisiä tyylejä lomakkeisiin, taulukoihin, nappeihin, luetteloihin ja kuviin. Bootstrapissa on valmiiksi erilaisia komponentteja kuten esimerkiksi täysin toimivat navigointipalkit, jota auttavat ja helpottavat verkkosivujen luontia. Bootstrapin ydin JavaScript lisää hyödyllistä koodia modaalien, kuva karusellien, hälytysten, ponnahdusikkunoiden ja pudotusvalikkojen luomiseen. (Rascia 2015.)




Kuva 15. Bootstrap ruudukko(grid) systeemi (Rascia 2015.)

## 3.8 Reactstrap

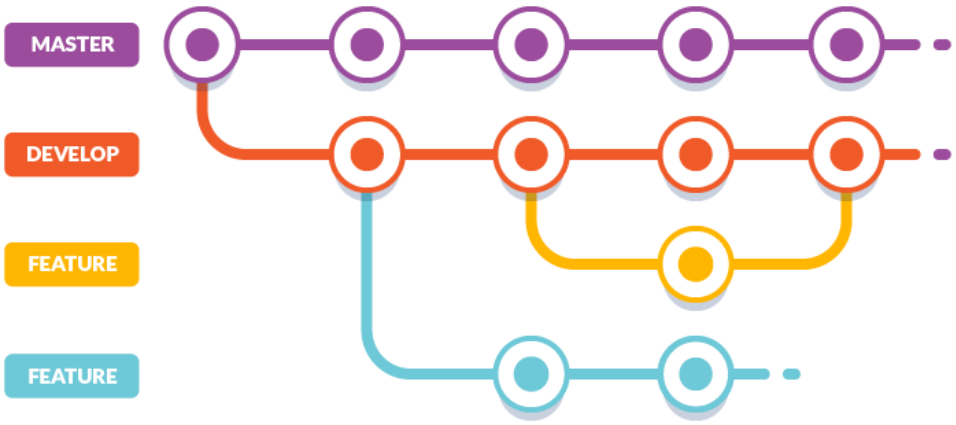
Reacstrap on kirjasto, joka sisältää React Bootstrap 4 -komponentit. Nämä komponentit ovat tehty juuri Reactia varten, huomioiden Reactin ominaisuudet. Reacstrap ei riipu jQuery- tai Bootstrap-JavaScriptistä, mutta muutamat komponentit tarvitsevat PopperJS eli react-popper paketin/kirjaston. (Reactstrap 2019.)

# 4 VERSIONHALLINTAOHJELMISTO

## 4.1 Git

Git on ohjelmisto, jonka avulla voit suorittaa versionhallintaa. Suuret ohjelmistoprojektit vaativat jonkinlaista ohjelmistoa seuraamaan kaikkia koodipohjaan tehtyjä muutoksia. Gitin avulla saadaan esimerkiksi selville, kuka on editoinut tiettyä tiedostoa, mitä tiedostosta on muutettu ja miten pääsee takaisin alkuperäiseen koodiin tarvittaessa. (Elder 2016.) Git on aktiivisesti ylläpidetty avoimen lähdekoodin hajautettu versionhallintaohjelmisto, jonka Linus Torvalds on kehittänyt vuonna 2005. Hajautettu järjestelmä helpottaa työntekoa, koska jokaisella kehittäjällä on lokaali repository koko ohjelmistosta. Tämä tarkoittaa, että koodia ei tallenneta vain keskuspalvelimeen, vaan koodi on myös kokonaisuudessaan kaikkien kehittäjien lokaalissa repositoryssä. Git on suunniteltu toimimaan mahdollisimman nopeasti ja tehokkaasti. (Sridhar 2018.)

Git:n tärkeimpiä käsitteitä ovat master ja branch. Master on pääkehityshaara, johon ei yleensä tehdä suoraan muutoksia. Tarkoituksena on tehdä kehityshaara(branch), joka valmiina ollessaan yhdistetään master haaraan. (Sridhar 2018) Kuvassa 16 nähdään esimerkki haarojen käytöstä: master ja develop ovat kaksi rinnakkaista pitkää haaraa. Tässä työnkulussa master haara on aina niin sanotusti tuotantovalmis ja develop on haara, johon lisätään kaikki ominaisuudet(feature) testausta varten ennen master haaraan yhdistämistä.




Kuva 16. Gitflow (Buddy 2016.)

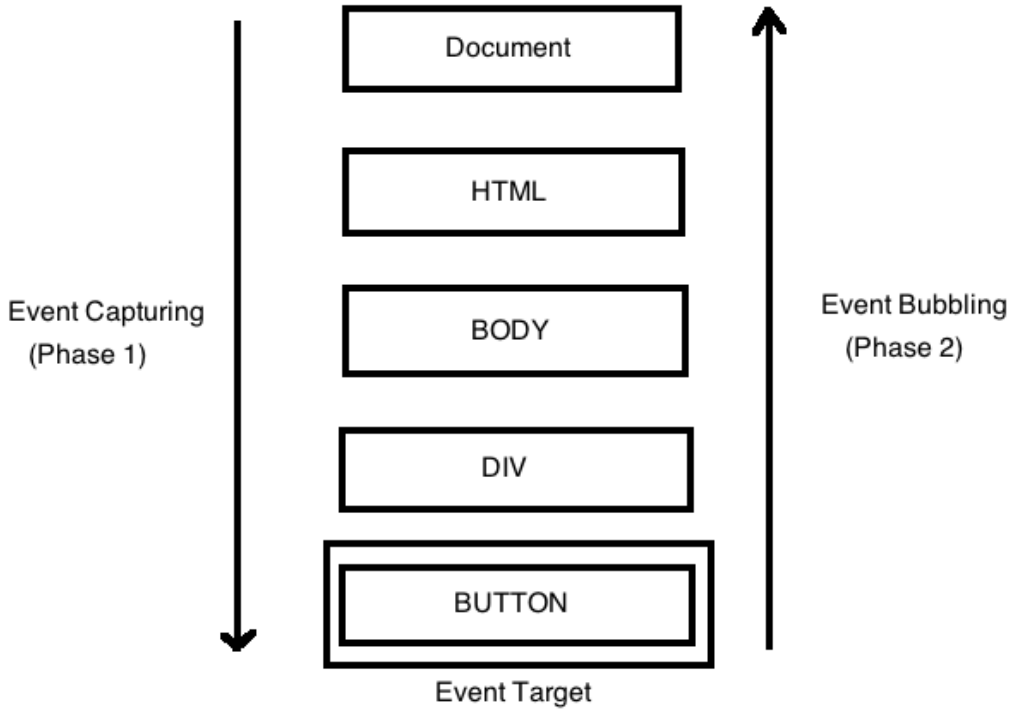
## 4.2 Bitbucket

Bitbucket on web-pohjainen Git-koodinhallintatyökalu. Bitbuketissa on paljon hyviä ominaisuuksia ja työkaluja, mutta ehkä tärkein niistä on se, että koodin pystyy tallentamaan turvallisesti turvalliseen paikkaa. Bitbucketin on kehittänyt Atlassian vuonna 2008. Mahdollisuus käyttää Bitbucketia Git-projekteissa tuli vuonna 2011. Bitbucket tarjoaa kaupallisia ja ilmaisia palveluja. Bitbucket integroituu hyvin Atlassianin muiden ohjelmien kanssa. (Vishwakarma 2016.)

# 5 TOTEUTUS

## 5.1 Testaus

Ennen varsinaisen toteutuksen aloittamista todettiin, että muutamia käyttöliittymään tulevia toiminnallisuuksia olisi hyvä testata. Suurimpana testauksen kohteena oli viivakoodinlukijan käsittely ja siihen mahdollisesti vaikuttava tapahtumien kupliminen (Kuva 17). Tarkoituksena oli testata, että viivakoodin skannaamisen jälkeen käsitelty viivakoodi ei kuplisi muihin elementteihin. Kuplauksen testaaminen oli tärkeää, koska haluttiin että viivakoodin skannaaminen ei tarvitsisi esimerkiksi input kentän fokusointia, vaan viivakoodin skannaamiseen riittäisi, että selainikkuna olisi auki. Kuplauksen varsinaiseen testaamiseen päästiin vasta, kun oli valittuna tapa, miten tarkalleen ottaen viivakoodinlukija käsitellään.




Kuva 17. Tapahtumien kulku kuplien ja kaapaten (Sharma 2017.)

Viivakoodinlukijan käsittelyyn mietittiin aluksi jo valmiin kirjaston käyttöä, mutta esimerkiksi valmiita npm-paketteja löytyi todella vähän. Myös näiden kirjastojen muuttaminen työn tarpeiden mukaiseksi olisi ollut hankalaa ja aikaa vievää. Niinpä

todettiin, että on parempi tehdä oma koodi käsittelemään viivakoodinlukijaa. Itse tehty käsittelijä on hyvä käyttöliittymän kehityksen kannalta, jos esimerkiksi viivakoodinlukijan käsittelyyn tarvitaan muutosta, niin on hyvä olla mahdollisuus muuttaa koodia mahdollisimman alhaisella tasolla.

Viivakoodinlukijan käsittelyä lähdettiin toteuttamaan testausmielessä. Testauksessa kokeiltiin erilaisia ratkaisuja viivakoodinlukijan käsittelyyn. Ratkaisu, johon päädyttiin, oli varsin looginen, viivakoodinlukijaa käsitellään kuten näppäimistöä ja selainikkunaan lisätään tapahtuma kuuntelija kuuntelemaan viivakoodinlukijan painalluksia. Viivakoodi pystytään käsittelemään, jos tiedetään viivakoodinlukijan etuliite(prefix) ja jälkiliite(suffix). Kun etu- ja jälkiliite on tiedossa, saadaan selville niiden välissä oleva viivakoodi (Kuva 18). Etu- ja jälkiliite riippuvat viivakoodinlukijasta, joissakin viivakoodinlukijoissa näitä liitteitä pystyy muuttamaan. Tässä työssä käytettiin Opticon OPR-3201 USB viivakoodinlukijaa (Kuva 19), jossa oletus etuliitteenä on Ctrl + B ja jälkiliitteenä Ctrl + C. Nämä oletusarvot muutetaan projektikohtaisesti, niin ettei käyttäjä voisi niitä vahingossa aktivoida.


Kuva 18. Etu- ja jälkiliite (SixBit 2019.)

Kuva 19. Opticon OPR-3201 USB viivakoodinlukija (Koskinen 2019.)

Kuvassa 20 nähdään testauksen yhteydessä tehty JavaScript-koodi, joka käsittelee viivakoodinlukijalla skannattua viivakoodia. Luodaan selainikkunalle (window) tapahtuman kuuntelija (EventListener), johon syötetään parametreina haluttu tapahtuma (”keydown”), joka suoritetaan tapahtuman yhteydessä ja tieto halutaanko tapahtumat etenevän kuplien (bubbling) tai kaapaten (capturing). Aluksi koodissa luodaan tapahtuman kuuntelija etuliitteelle, joka käydään läpi funktiossa allkeysPre. Jos kyseinen etuliite löytyy, luodaan uusi kuuntelija jälkiliitteelle. Funktiossa allkeysSuf käydään lävitse jälkiliite. Jos jälkiliite löytyy skannatusta viivakoodista, viivakoodi tulostetaan tekstikentälle. Ennen viivakoodin tulostamista pitää muistaa, että tapahtuma kuuntelija huomio jokaisen näppäimen painalluksen: esimerkiksi Shift painallus isoissa kirjaimissa, joka koodissa poistetaan funktiolla move.

```
const id = ['txtfield2', 'txtfield3', 'txtfield4', 'txtfield5'];
var barcode = [];
var suf = [];
var pre = [];
const prefix = ['Control', 'b'];
const suffix = ['Control', 'c'];
var txt = document.getElementById('txtfield2');
var listener = document.getElementById('txtfield');

disable('txtfield');
for(var i=0; i < id.length; i++) {
  disable(id[i]);
}

window.addEventListener('keydown', allkeysPre, true);

function allkeysPre(event){
  pre.push(event.key);
  pre.splice(0, pre.length - prefix.length);

  if(pre.join("") === prefix.join("")) {
    remove(window, allkeysPre);
    add(window, allkeysSuf);
  }
}

function allkeysSuf(event){
  suf.push(event.key);
  suf.splice(0, suf.length - suffix.length);

  //If suffix is found.
  if(suf.join("") === suffix.join("")) {
    //Move index/element.
    move(barcode, 'Shift');
    for(var i = 0; i < suf.length; i++) {
      move(barcode, suf[i]);
    }

    document.getElementById('txtfield').value = barcode.join("");

    for(var i = 0; i < id.length; i++) {
      document.getElementById(id[i]).value = id[i];
    }

    remove(window, allkeysSuf);

    //Array reset.
    pre.length = 0;
    suf.length = 0;
    barcode.length = 0;

    for(var i=0; i < id.length; i++){
      enable(id[i]);
    }

  }
  else {
    barcode.push(event.key);
  }
}
function remove(element, func) {
  element.removeEventListener('keydown', func, true);
}
function add(element, func) {
  element.addEventListener('keydown', func, true);
}
function disable(element) {
  document.getElementById(element).disabled = true;
}
function enable(element) {
  document.getElementById(element).disabled = false;
}
function move(arr, val) {
  var j = 0;
  for (var i = 0, l = arr.length; i < l; i++) {
    if (arr[i] !== val) {
      arr[j++] = arr[i];
    }
  }
  arr.length = j;
}
```

Kuva 20. Viivakoodinlukijan käsittely koodi (Koskinen 2019.)

Testauksen tulos osoitti sen, että tämä tavoite pystytään toteuttamaan. Työn vaatimukset ja toiminnallisuudet voidaan toteuttaa web-toteutuksena. Testauksen ohella tehtiin ensimmäinen versio viivakoodinlukijan käsittelijästä, ja käsittelijää tehdessä huomattiin, että tapahtuman kuuntelija voidaan valita etenemään joko kuplien tai kaapaten. Tapahtumien mahdollinen kupliminen ei siis olisi työn esteenä. Toiminnallisuudet saatiin testattua onnistuneesti, joten käyttöliittymän toteutus voitiin aloittaa.

## 5.2 React-kirjaston asennus

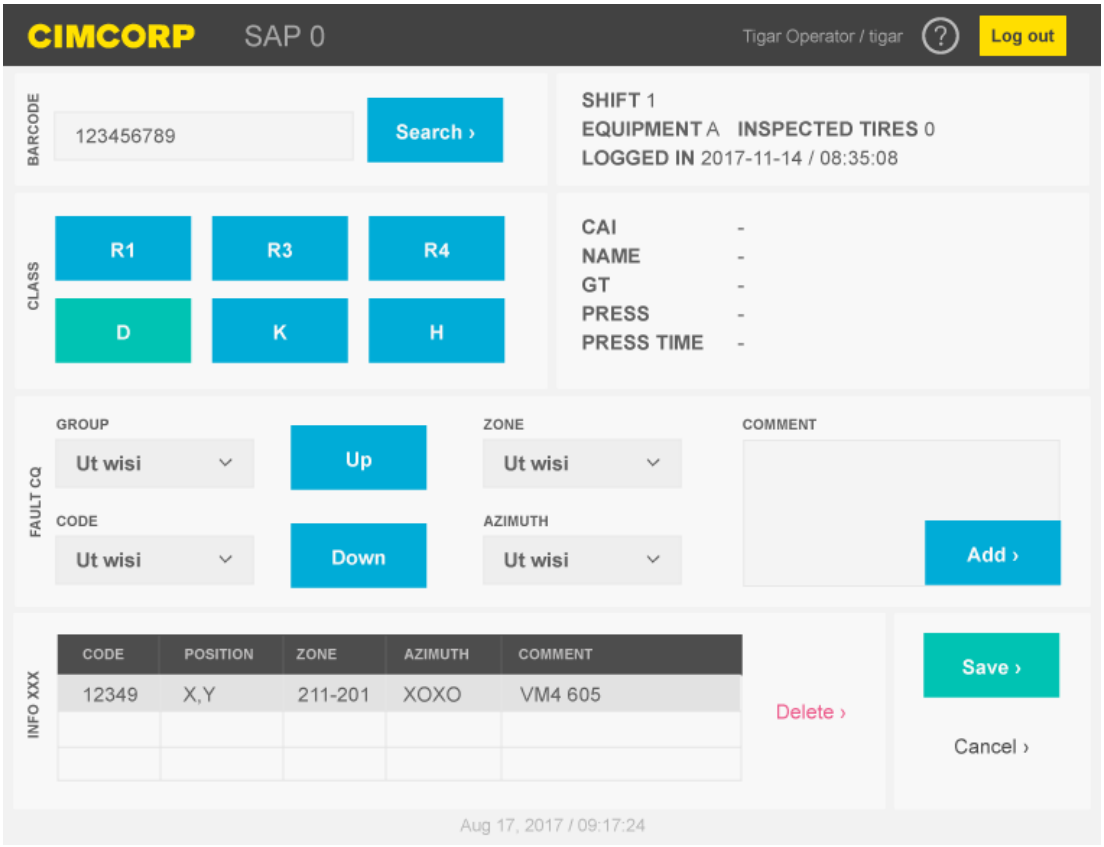
React-kirjaston asentamiseen tarvitaan paketinhallintaohjelma. Paketinhallintaohjelmista valinta kohdistui npm eli Node.js package manageriin. Npm tulee Node.js asennuksen mukana, joten Node.js tulee ladata ensin. Node.js:n lataamisen ja asentamisen jälkeen React-kirjaston eli create-react-app paketin asentaminen tapahtuu komentokehotteen kautta komennolla ”npm init create-app my-app”, joka luo my-app nimisen kansion nykyisen kansion sisälle. Paketin asentamisen jälkeen siirrytään juuri luotuun kansioon, jossa käynnistetään sovellus kehitystilassa komennolla ”npm start”. Kyseinen komento avaa sovelluksen selaimeen ”http://localhost:3000” osoitteessa. Kehitystilaa on järkevä pitää koko ajan päällä, jotta näkee tehdyt muutokset selaimessa samaan aikaan kun koodi muuttuu.

Tässä työssä käyttöliittymä tehtiin Cimcorp Oy:llä valmiiksi kehitysvaiheessa olevan järjestelmän ”tyhjälle” sivulle. Create-react-app paketin asennuksen jälkeen kansiosta poistettiin kaikki paketin mukana tuleva, jonka jälkeen kansioon siirrettiin kyseinen järjestelmä. Tämän jälkeen järjestelmän kansioon tehtiin QStationApp kansio, jonka sisälle alettiin luoda kyseistä käyttöliittymää.

Asennuksen jälkeen luotiin oma kehitys branch käyttöliittymälle, ja luotiin Bitbuckettiin oma repository. Koska tässä työssä oli vain yksi kehittäjä, ei ollut tarvetta kuin yhdelle branchille.

## 5.3 Komponenttien luonti

Komponenttien suunnittelu aloitettiin käyttöliittymän luonnoksesta (Kuva 21). Luonnos helpotti käyttöliittymän suunnittelua, koska tulevat komponentit voitiin määritellä jo etukäteen. Ideaalitilanteessa kaikki mahdollinen olisi komponentteja. Tässä käyttöliittymässä komponentit ovat tärkeitä, koska käyttöliittymän mahdollinen tulevaisuuden kehitys ja jokaisessa projektissa kyseinen käyttöliittymä voi olla erilainen. Komponenttien suunnittelussa huomattiin, että käyttöliittymässä voidaan hyvin käyttää Bootstrapin ruudukko(grid) systeemiä. Niinpä käyttöliittymä ajateltiin, neljänä rivinä(row) ja kahtena sarakkeena(column) jokaisen rivin sisällä. Päätettiin, että jokainen sarake olisi oma komponentti ja näiden komponenttien sisälle tulisi valmiita Reactstrapin Boostrap komponentteja: näin ollen komponentteja tehtäisiin kahdeksan kappaletta. Yläpalkki (TopBar) ja alapalkki (DevFooter) komponentit löytyivät jo entuudestaan, joten niille ei tarvinnut tehdä komponentteja.




Kuva 21. Tarkastusaseman käyttöliittymän luonnoskuva (Cimcorp Oy 2018.)

Kuvassa 22 nähdään QStationApp.js tiedoston koodi, jossa käytetään tehtyjä komponentteja. Koodissa näkyy, miten komponentit määritellään ja asetetaan. Bootstrap ruudukon rivit määritetään tässä, mutta sarakkeet jokaisen komponentin koodissa.

```
export class QStationApp extends Component {
    render() {
        if (this.props.initializing || this.props.initializationError) {
            return <LoadingOverlay
                isLoading={this.props.initializing}
                errorDetail={this.props.initializationError} />;
        } else return (
            <div>
                <TopBar menuEnabled={false} />
                <div className="MiddleContent">
                    <NotificationContainer />
                    <TitledContainer title={"QStation"}>
                        <Row>
                            <Barcode />
                            <Shift />
                        </Row>
                        <Row>
                            <QualityClass />
                            <TireInfo />
                        </Row>
                        <Row className="faultcomment">
                            {this.props.drawFault ?
                            <Fault /> : null
                            }
                            <CommentArea />
                        </Row>
                        <Row>
                            <FaultInfo />
                            <SaveCancel />
                        </Row>
                    </TitledContainer>
                    <DevFooter />
                </div>
            </div>
        );
    }
};

QStationApp.defaultProps = {
    initializing: false,
    drawFault: true
}

QStationApp.propTypes = {
    initializing: PropTypes.bool.isRequired,
    initializationError: PropTypes.string,
    drawFault: PropTypes.bool
}

export default withRouter(connect(
    function (state) {
        const { fetching, error } = state.featureList;
        return {
            initializing: fetching,
            initializationError: error
        };
    },
    function (dispatch) {
        return {};
    }
)(QStationApp));
```

Kuva 22. QStationApp.js komponenttien ”root” koodi (Koskinen 2019.)

### 5.3.1 Barcode.js komponentti

Tässä komponentissa luodaan ensimmäisen rivin ensimmäinen sarake. Sarakkeen sisälle luodaan lomake, jonka sisälle luodaan input tekstikenttä ja Search-nappula käyttäen React Boostrap 4 komponentteja. Komponentissa käsitellään viivakoodinlukijalla skannattu tai tekstikenttään kirjoitettu viivakoodi, jos viivakoodi kirjoitetaan, Search-nappulaa täytyy erikseen painaa. Search-napin painallus hakee viivakoodin avulla renkaan datan. Input tekstikentässä suoritetaan myös alfanumeerinen tarkastelu viivakoodille, joten tekstikenttään ei pysty kirjoittamaan tai skannaamaan ei-alfanumeerisia kirjaimia/numeroita. Search-nappulan painalluksen voi myös suorittaa Enter-näppäimellä. (Kuva 23)

```
export class Barcode extends Component {
    constructor(props, context) {
        super(props, context);

        this.inputRef = React.createRef();
    }

    componentDidUpdate(prevProps) {
        if(this.props.readWithBarcodeScanner.active) {
            this.props.addChar(this.props.barcode)
            this.props.setReadWithBarcodeScanner(false)
        }
        if(this.props.cancelbutton.active) {
            this.props.resetAddBarcode()
            this.inputRef.focus()
        }
        if(this.props.blur.active) {
            this.inputRef.blur()
        }
     }

    componentDidMount() {
        this.inputRef.focus()
    }

    handleClick = () => {
        this.props.setBarcode(this.props.addBarcode, true)
        this.props.fetchData(this.props.addBarcode);
    }

    onInputChange = (event) => {
        let reg = new RegExp(/^[a-z0-9]+$/i)

        if(event.target.value.length === 0) {
            this.props.resetAddBarcode()
        }

        if(reg.test(event.target.value)) {
            this.props.addChar(event.target.value)
        }
        console.log(event.target.value)
    }

    handleKeyPress = (target) => {
        if(target.key === "Enter") {
            target.preventDefault()
            this.handleClick()
            this.props.setBlurActive(true)
        }
    }

    render() {
        return (
            <Col md="7" className="barcode">
            <Label>Barcode</Label>
                <Form inline>
                <FormGroup>
                    <Input
                        innerRef={input => (this.inputRef = input)}
                        type="text"
                        value={this.props.addBarcode}
                        onChange={this.onInputChange}
                        onKeyPress={this.handleKeyPress}/>
                </FormGroup>
                <Button className="searchBtn" onClick={this.handleClick} color="primary" size="lg">Search ></Button>
                </Form>
            </Col>
        );
    }
};
```

Kuva 23. Barcode.js komponentti (Koskinen 2019.)

### 5.3.2 Shift.js komponentti

Shift komponentti oli näistä kaikista kahdeksasta komponentista vähiten panostettu, koska tähän komponenttiin vaikuttaa eniten operaattorin kirjautumisessa saadut tiedot ja tässä työssä ei ollut tarkoituksena tehdä kirjautumista. Kirjautumissivu on Cimcorp Oy:llä jo kehitys vaiheessa, joten kun se saadaan valmiiksi, lisätään se tähän käyttöliittymään. Tässä komponentissa luodaan ensimmäisen rivin toinen sarake. Sarakkeen sisällä luodaan kovakoodatut tekstit ja kellonaika. (Kuva 24)

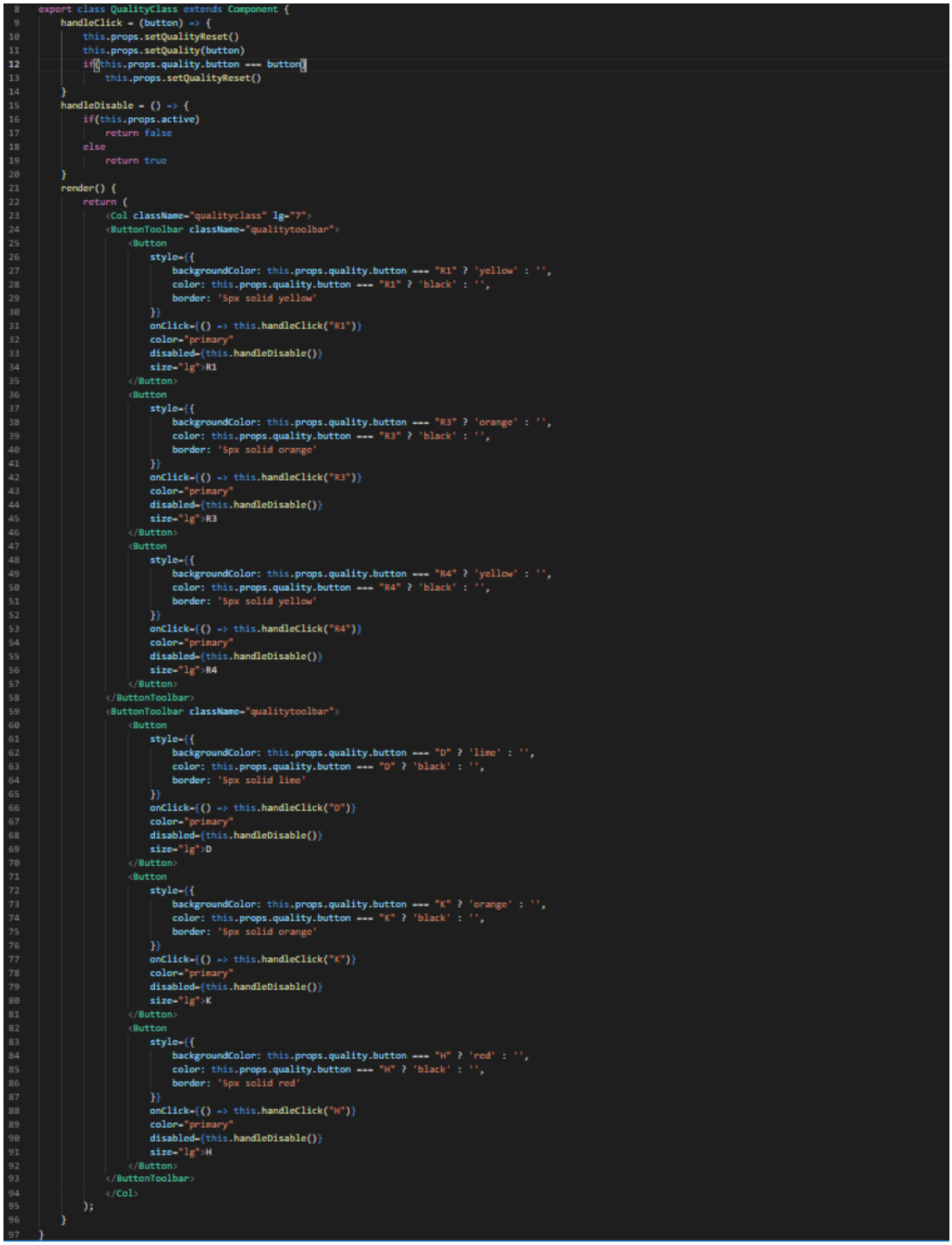
```
1  import React, { Component } from 'react';
2
3  import { Col  } from 'reactstrap';
4  import './Shift.scss';
5
6  const d = new Date();
7
8  export class Shift extends Component {
9
10     render() {
11         return (
12                 <Col md="5" className="shift">
13                     <p>SHIFT: 1</p>
14                     <p>
15                         EQUIPMENT: A
16                         INSPECTED TIRES: 0
17                     </p>
18                     <p>LOGGED IN: {d.toLocaleString()}</p>
19                 </Col>
20         );
21     }
22 }
23
24 export default Shift;
```

Kuva 24. Shift.js komponentti (Koskinen 2019.)

### 5.3.3 QualityClass.js komponentti

Tässä komponentissa luodaan toisen rivin ensimmäinen sarake. Sarakkeen sisällä luodaan nappuloille kaksi nappiryhmää (ButtonToolbar), jotka yhdistävät työkalupalkin napit niin sanotusti yhteiseksi ryhmäksi. Nappulat ja työkalupalkit luodaan React Bootstrap 4 komponentteja käyttäen. Nappeja luodaan kuusi kappaletta. Kun nappia painetaan, nappi värittyy kokonaan samalla värillä millä nappi on rajattu. Esimerkiksi nappi R1 on rajattu keltaisella, kun sitä painaa koko nappi muuttuu keltaiseksi. Napin painallus muuttaa Redux storessa olevan quality tilan laittaen tälle arvoksi painetun napin nimen, esimerkiksi R1. Napin painalluksen eli laatuluokan valinnan voi myös poistaa painamalla nappia uudestaan, jolloin tilan arvo muuttuu

takaisin oletusarvoksi. Napit aktivoituvat vasta, kun viivakoodi on skannattu tai kirjoitettu, ja data on onnistuneesti haettu. (Kuva 25)



```
export class QualityClass extends Component {
    handleClick = (button) => {
        this.props.setQualityReset()
        this.props.setQuality(button)
        if(this.props.quality.button === button)
            this.props.setQualityReset()
    }
    handleDisable = () => {
        if(this.props.active)
            return false
        else
            return true
    }
    render() {
        return (
            <Col className="qualityclass" lg="7">
            <ButtonToolbar className="qualitytoolbar">
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "R1" ? 'yellow' : '',
                        color: this.props.quality.button === "R1" ? 'black' : '',
                        border: '5px solid yellow'
                    }}
                    onClick={() => this.handleClick("R1")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">R1
                </Button>
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "R3" ? 'orange' : '',
                        color: this.props.quality.button === "R3" ? 'black' : '',
                        border: '5px solid orange'
                    }}
                    onClick={() => this.handleClick("R3")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">R3
                </Button>
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "R4" ? 'yellow' : '',
                        color: this.props.quality.button === "R4" ? 'black' : '',
                        border: '5px solid yellow'
                    }}
                    onClick={() => this.handleClick("R4")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">R4
                </Button>
            </ButtonToolbar>
            <ButtonToolbar className="qualitytoolbar">
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "D" ? 'lime' : '',
                        color: this.props.quality.button === "D" ? 'black' : '',
                        border: '5px solid lime'
                    }}
                    onClick={() => this.handleClick("D")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">D
                </Button>
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "K" ? 'orange' : '',
                        color: this.props.quality.button === "K" ? 'black' : '',
                        border: '5px solid orange'
                    }}
                    onClick={() => this.handleClick("K")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">K
                </Button>
                <Button
                    style={{
                        backgroundColor: this.props.quality.button === "H" ? 'red' : '',
                        color: this.props.quality.button === "H" ? 'black' : '',
                        border: '5px solid red'
                    }}
                    onClick={() => this.handleClick("H")}
                    color="primary"
                    disabled={this.handleDisable()}
                    size="lg">H
                </Button>
            </ButtonToolbar>
            </Col>
        );
    }
}
```

Kuva 25. QualityClass.js komponentti (Koskinen 2019.)

### 5.3.4 TireInfo.js komponentti

TireInfo komponentissa luodaan toisen rivin toinen sarake. Sarakkeen sisällä luodaan taulukko React Bootstrap 4 komponenttia käyttäen. Taulukon data tulee Redux storesta, fetchdata tilasta. Fetchdata tila päivittyy, jos skannatun tai kirjoitetun viivakoodin data löytyy. Kun data löytyy, näytetään renkaan tiedot taulukossa. Tällä hetkellä data haetaan JSON-tiedostosta, mutta tässä työssä käytettiin vain testidataa, joten datan hakeminen voi vielä muuttua. (Kuva 26)

```
export class TireInfo extends Component {
    render() {
        return(
            <Col lg="5" className="tireinfo">
            <Table hover borderless>
                <tbody>
                    <tr>
                    <th scope="row">CAI</th>
                    <td>{this.props.fetchdata.barcode}</td>
                    </tr>
                    <tr>
                    <th scope="row">NAME</th>
                    <td>{this.props.fetchdata.name}</td>
                    </tr>
                    <tr>
                    <th scope="row">GT</th>
                    <td>{this.props.fetchdata.gt}</td>
                    </tr>
                    <tr>
                    <th scope="row">PRESS</th>
                    <td>{this.props.fetchdata.press}</td>
                    </tr>
                    <tr>
                    <th scope="row">PRESS TIME</th>
                    <td>{this.props.fetchdata.pressTime}</td>
                    </tr>
                </tbody>
            </Table>
            </Col>
        );
    }
}

export default connect(
    function (state, ownProps) {
        return {
            barcode: state.qcontrol.inspectable || '',
            fetchdata: state.fetchdata || ''

        };
    },
    function (dispatch, ownProps) {
        return {

        };
    }
)(TireInfo);
```

Kuva 26. TireInfo.js komponentti (Koskinen 2019.)

### 5.3.5 Fault.js komponentti

Fault komponentissa luodaan kolmannen rivin ensimmäinen sarake. Sarakkeen sisälle luodaan kaksi nappiryhmää, ja näiden sisälle kaksi pudotusvalikkonappulaa ja yksi normaali nappula. Periaate on se, että jokaisella valikkonapilla on oma tilansa ja näillä kahdella normaalilla napilla on yhteinen oma tila. Nappien tilan muuttuessa, tila päivittyy Redux storessa. Napit aktivoituvat vasta, kun viivakoodi on skannattu tai haettu Search-nappulalla. (Kuva 27)

```
72      render() {
73          return (
74              <Col lg="8" className="fault">
75              <ButtonToolbar className="buttontoolbar">
76                  <UncontrolledDropdown>
77                      <DropdownToggle id="toggle" caret size="lg" color="default" disabled={this.handleDisable()}>
78                          {this.props.group.title}
79                      </DropdownToggle>
80                      <DropdownMenu
81                          id="dropdownButton"
82                          modifiers={{
83                              setMaxHeight: {
84                                enabled: true,
85                                order: 890,
86                                fn: (data) => {
87                                  return {
88                                    ...data,
89                                    styles: {
90                                      ...data.styles,
91                                      overflow: 'auto',
92                                      maxHeight: 200,
93                                    },
94                                  };
95                                },
96                              },
97                            }}>
98                          <DropdownItem header>Group</DropdownItem>
99                          <DropdownItem onClick={this.handleSelectGroup}>10</DropdownItem>
100                         <DropdownItem onClick={this.handleSelectGroup}>30</DropdownItem>
101                         <DropdownItem onClick={this.handleSelectGroup}>40</DropdownItem>
102                         <DropdownItem onClick={this.handleSelectGroup}>50</DropdownItem>
103                         <DropdownItem onClick={this.handleSelectGroup}>60</DropdownItem>
104                         <DropdownItem onClick={this.handleSelectGroup}>70</DropdownItem>
105                         <DropdownItem onClick={this.handleSelectGroup}>80</DropdownItem>
106                         <DropdownItem onClick={this.handleSelectGroup}>90</DropdownItem>
107                         <DropdownItem onClick={this.handleSelectGroup}>200</DropdownItem>
108                     </DropdownMenu>
109                 </UncontrolledDropdown>
110                 <Button
111                     onClick={() => this.handlePositionClick("UP")}
112                     style={{
113                         backgroundColor: this.props.position.fault === "UP" ? 'Aqua' : '',
114                         color: this.props.position.fault === "UP"  ? 'black' : '',
115                         border: '5px solid Aqua'
116                     }}
117                     id="positionButton"
118                     color="primary"
119                     disabled={this.handleDisable()}
120                     size="lg">UP
121                 </Button>
122                 <UncontrolledDropdown>
123                     <DropdownToggle id="toggle" caret size="lg" color="default" disabled={this.handleDisable()}>
124                         {this.props.zone.title}
125                     </DropdownToggle>
126                     <DropdownMenu
127                         id="dropdownButton"
128                         modifiers={{
129                             setMaxHeight: {
130                               enabled: true,
131                               order: 890,
132                               fn: (data) => {
133                                 return {
134                                   ...data,
135                                   styles: {
136                                     ...data.styles,
137                                     overflow: 'auto',
138                                     maxHeight: 200,
139                                   },
140                                 };
141                               },
142                             },
143                           }}>
144                         <DropdownItem header>Zone</DropdownItem>
145                         <DropdownItem onClick={this.handleSelectZone}>11</DropdownItem>
146                         <DropdownItem onClick={this.handleSelectZone}>12</DropdownItem>
147                         <DropdownItem onClick={this.handleSelectZone}>13</DropdownItem>
148                         <DropdownItem onClick={this.handleSelectZone}>14</DropdownItem>
149                         <DropdownItem onClick={this.handleSelectZone}>15</DropdownItem>
150                         <DropdownItem onClick={this.handleSelectZone}>21</DropdownItem>
151                         <DropdownItem onClick={this.handleSelectZone}>22</DropdownItem>
152                         <DropdownItem onClick={this.handleSelectZone}>27</DropdownItem>
153                         <DropdownItem onClick={this.handleSelectZone}>28</DropdownItem>
154                         <DropdownItem onClick={this.handleSelectZone}>31</DropdownItem>
155                         <DropdownItem onClick={this.handleSelectZone}>32</DropdownItem>
156                         <DropdownItem onClick={this.handleSelectZone}>41</DropdownItem>
157                         <DropdownItem onClick={this.handleSelectZone}>42</DropdownItem>
158                     </DropdownMenu>
159                 </UncontrolledDropdown>
160             </ButtonToolbar>
```

Kuva 27. Fault.js komponentin toinen nappiryhmä (Koskinen 2019.)

Tässä komponentissa Code valikkonappulan sisältö riippuu Group valikkonappulan valinnasta, joten tähän tarvittiin erilainen menettelytapa. Luotiin Groups funktio, joka lisää Code valikkonappulaan sisältöä riippuen Group valikkonappulan valinnasta (Kuva 28). Funktiossa käytetään apuna codes.json tiedostoa, jonka sisällä on Code valikkonappulan data (Kuva 29.) Funktio etsii ensin Group valikkonappulasta valitun valinnan JSON-objekteista. Esimerkiksi, jos valinta on ”10”, funktio etsii JSON-tiedostosta objektin, jossa Group arvo on ”10”. Jos funktio löytää arvon se etsii ja muuttaa (map) kyseisestä objektista löytyvät code datat DropdownItem komponenteiksi.

```
59    Groups = () => {
60        const groupFound = codes.find(codes => codes.group === this.props.group.fault)
61
62        if(groupFound) {
63            return groupFound.code.map((code, index) =>
64                <DropdownItem key={index} onClick={this.handleSelectCode}>{code}</DropdownItem>
65            )
66        }
67        else {
68            return null;
69        }
70    }
```

Kuva 28. Fault.js komponentin Groups funktio (Koskinen 2019.)

```
1  [
2      {"group": "10", "code": [11.1, 11.2, 13.1, 15.1, 16.1, 17.1, 19.1]},
3      {"group": "30", "code": [31.1, 32.2, 33.1, 35.1, 35.2, 35.3, 36.1, 38.1, 38.2]},
4      {"group": "40", "code": [42.1, 42.2, 43.4, 44.1, 44.2, 45.1, 45.2, 45.3, 46.1, 47.1]},
5      {"group": "50", "code": [51.4, 52.1, 52.2, 52.3, 54.1]},
6      {"group": "60", "code": [61.1, 61.2, 61.3, 62.1, 62.2, 65.2, 66.1, 66.3, 66.4, 69.1]},
7      {"group": "70", "code": [71.1, 72.1, 74.1, 74.2, 74.4, 75.1, 75.2, 79.1]},
8      {"group": "80", "code": [81.1, 82.1, 82.3, 83.1, 84.1]},
9      {"group": "90", "code": [91.1, 91.2, 91.3, 93.1, 94.1, 95.1, 95.2, 98.1]},
10     {"group": "200", "code": [204.1, 216.1, 225.1]}
11 ]
12
```

Kuva 29. Funktiossa Groups käytetty JSON-tiedosto codes.json (Koskinen 2019.)

### 5.3.6 CommentArea.js komponentti

Tässä komponentissa luodaan kolmannen rivin toinen sarake. Sarakkeen sisällä luodaan lomake, jonka sisällä luodaan input tekstikenttä kommentille ja Add-nappula. Tämän komponentin tarkoituksena on lisätä kaikki edellisten komponenttien storen tilassa oleva data omaan allinfo tilaan. Kommentti lisätään aluksi vain komponentin omaan tilaan, mutta kun Add-nappia painetaan, se lähtee myös Redux storessa olevaan allinfo tilaan. CommentArea komponentissa tarkastellaan myös, että onko edellisissä komponenteissa annettu data oikeaa, ja katsotaan, onko allinfo tilassa jo mak-

simi määrä annettuja laatuvirheitä eli kolme kappaletta. Tässäkin komponentissa koko lomake aktivoituu vasta, kun viivakoodi on skannattu tai kirjoitettu.

```
export class CommentArea extends Component {
    constructor(props, context) {
        super(props, context);

        this.state = {
            value: ""
        };

        this.inputref = React.createRef();
    }
    componentDidUpdate(prevProps) {
        if(this.props.addbutton.active) {
            this.handleClick();
            this.props.setAddButtonActive(false)
        }

        if(this.props.cancelbutton.active) {
            this.setState({value: ""})
            this.props.setCancelButtonActive(false)
        }

        if(this.props.blur.active) {
            this.inputref.blur();
            this.props.setBlurActive(false)
        }
      }
    handleChange = (event) => {
        this.setState({ value: event.target.value})
    }
    handleClick = () => {
        if(this.props.allinfo.length < 3 && this.props.azimuth.fault !== null && this.props.quality.button !== null &&
            this.props.code.fault !== null && this.state.value !== "" && this.props.group.fault !== null &&
            this.props.zone.fault !== null && this.props.position.fault !== null
        ) {
            this.props.setAllInfo(
                this.props.quality.button, this.props.group.fault,
                this.props.code.fault, this.props.position.fault,
                this.props.zone.fault, this.props.azimuth.fault,
                this.state.value
            )
            this.setState({value: ""})
            this.props.resetAdd()
            this.props.setAddButtonActive(false)
        }
    }
    handleDisable = () => {
        if(this.props.active)
            return false
        else
            return true
    }
    render() {
        return (
            <Col lg="4" className="CommentArea">
                <Form>
                    <FormGroup>
                        <Label for="exampleText">Comment</Label>
                        <Input type="textarea" name="text" id="exampleText" disabled={this.handleDisable()}
                        value={this.state.value} onChange={this.handleChange} innerRef={input => (this.inputref = input)} />
                        <Button onClick={this.handleClick} className="AddButton" color="primary" disabled={this.handleDisable()} size="lg">Add ></Button>
                    </FormGroup>
                </Form>
            </Col>
        );
    }
}
```

Kuva 30. CommentArea.js komponentti (Koskinen 2019.)

### 5.3.7 FaultInfo.js komponentti

FaultInfo komponentissa luodaan neljännen rivin ensimmäinen sarake. Sarakkeessa luodaan ReactTable-taulukko käyttäen ReactTable paketin komponenttia, ja Delete-nappula käyttäen React Bootstrap 4 komponenttia. Taulukko ottaa datansa allinfo tilasta, johon voidaan lisätä maksimissaan kolme laatuvirheinfoa, ja tämä data siis lisätään painamalla edellisen CommentArea komponentin Add-nappia. Taulukosta voidaan poistaa haluttuja laatuvirheitä. Tämä tapahtuu valitsemalla ensin laatuvirheet

taulukosta ja sitten painamalla Delete-nappia, varsinaista poistoa ei pysty peruuttamaan, mutta valinnan pystyy. Valitut rivit taulukossa korostetaan värillä, joten ne on helpompi huomata. (Kuva 31)

```
export class FaultInfo extends Component {
    constructor(props, context) {
        super(props, context);
        this.state = {
            selected: null,
        }
    }
    deleteRow = () => {
        if(this.state.selected !== null)
            this.props.deleteFaultInfos(this.state.selected)
            this.setState({selected: null})
    }
    handleDisable = () => {
        if(this.props.active)
            return false
        else
            return true
    }
    render() {
        const data = this.props.allinfo
        const columns = [{Header: 'Code', accessor: 'code'},
                         {Header: 'Position', accessor: 'position'},
                         {Header: 'Zone', accessor: 'zone'},
                         {Header: 'Azimuth', accessor: 'azimuth'},
                         {Header: 'Comment', accessor: 'comment'}]
        return (
            <Col lg="10" className="faultinfo">
            <Button className="DeleteButton" onClick={this.deleteRow} color="link" disabled={this.handleDisable()} size="lg">Delete ></Button>
                <ReactTable
                    data={data}
                    columns={columns}
                    minRows={3}
                    showPagination={false}
                    sortable={false}
                    resizable={false}
                    className='-striped -highlight'
                    getTrProps={(state, rowInfo) => {
                        if(typeof rowInfo !== "undefined") {
                            return {
                                onClick: (e, handleOriginal) => {
                                    if(this.state.selected !== rowInfo.index)
                                        this.setState({selected: rowInfo.index});
                                    else
                                        this.setState({selected: null})

                                    if(handleOriginal)
                                        handleOriginal()
                                },
                                style: {
                                    background: rowInfo.index === this.state.selected ? '#00afec' : '',
                                    color: rowInfo.index === this.state.selected ? 'white' : '',
                                },
                            }
                        }
                        else {
                            return {
                                onClick: (e, handleOriginal) => {
                                    if(handleOriginal)
                                        handleOriginal()
                                },
                                style: {
                                    background: 'white',
                                    color: 'black'
                                },
                            }
                        }
                    }
                }
                />
            </Col>
        );
    }
}
```

Kuva 31. FaultInfo.js komponentti (Koskinen 2019.)

Taulukon rivien poistaminen tapahtuu tilassa olevan datan poistamisella. Koska tilan muuttaminen reducerissa ei ole suositeltavaa, täytyy poistaminen tehdä slice funktiolla. Poistamista varten tehdään kopio tilasta, joten sitä voidaan turvallisesti muuttaa.

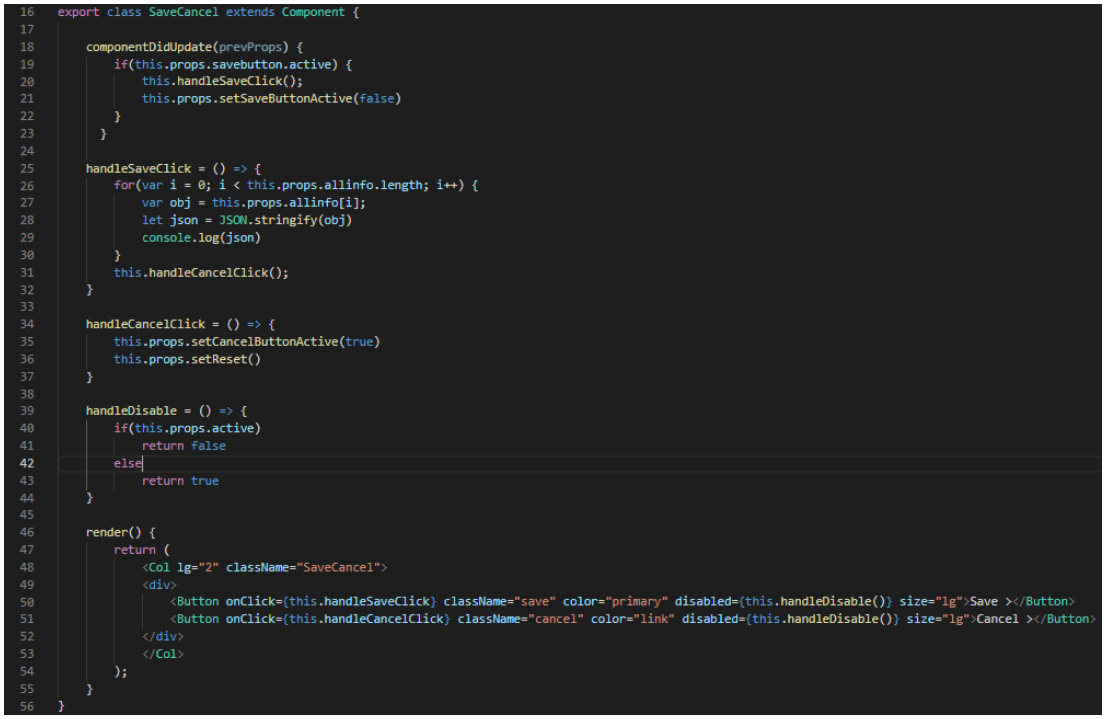
Kuvassa 32 nähdään kyseinen poisto, kun allinfo reducerille tulee Delete-actioni. Poiston jälkeen palautetaan uusi taulukko Redux storen allinfo tilaan.

```
247  function allinfo(state = qcontrolAllInfoDefaultState, action) {
248      switch(action.type) {
249          case Q_CONTROL_ALL_INFO:
250          let obj = {
251                  button: action.button,
252                  group: action.group,
253                  code: action.code,
254                  position: action.position,
255                  zone: action.zon any
256                  azimuth: action.azimuth,
257                  comment: action.comment
258          }
259          let newArr = state.concat(obj)
260              return newArr
261          case Q_CONTROL_DELETE_FAULT_INFOS:
262          let newArr2 = [
263              ...state.slice(0, action.index),
264              ...state.slice(action.index + 1)
265          ]
266              return newArr2
267          case Q_CONTROL_RESET_CANCEL:
268              return qcontrolAllInfoDefaultState
269          default:
270              return state;
271      }
272  }
```

Kuva 32. ReactTable-taulukon datan allinfo reducer (Koskinen 2019.)

### 5.3.8 SaveCancel.js komponentti

Tässä komponentissa luodaan neljännen rivin toinen sarake. Sarakkeessa luodaan kaksi nappulaa, Save- ja Cancel-nappula. Save-nappulan tarkoituksena on jäsentää allinfo tilassa oleva data JSON-objekti muotoon, koska tulevaisuudessa data on tarkoitus tallentaa tässä muodossa tietokantaan. Nyt tällä hetkellä data vain jäsennetään JSON-muotoon ja tulostetaan konsoliin. Cancel-nappulan tarkoituksena on resetoida kaikki Redux storessa kentissä ja valintana oleva data. Save- ja Cancel nappuloiden toimintoihin ei tarvita storeen tallennettua tilaa, vaan molemmat käyttävät jo olemassa olevia tiloja. Kaikissa reducereissa on Reset-actioni, mikä resetoi kaikkien komponenttien tilat oletusarvoihin. Datan tallentamiseen ei tarvita erillistä actionia, vaan data otetaan storesta ja muokataan sitä suoraan.

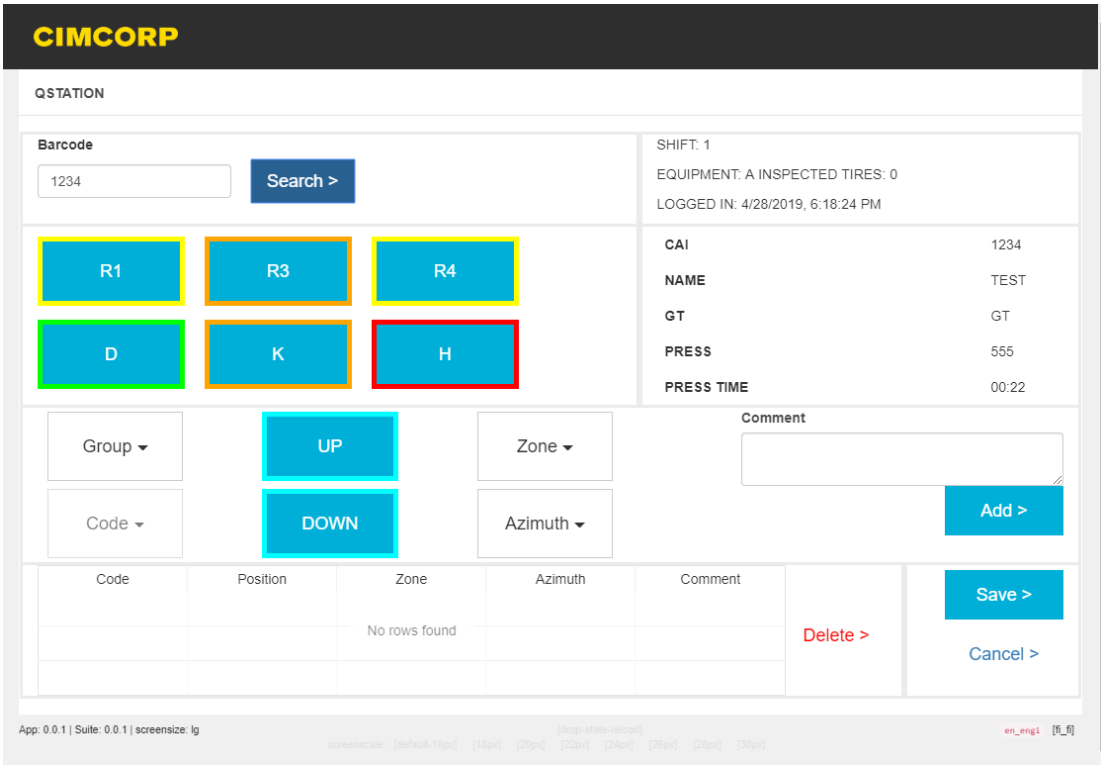


```
export class SaveCancel extends Component {

    componentDidUpdate(prevProps) {
        if(this.props.savebutton.active) {
            this.handleSaveClick();
            this.props.setSaveButtonActive(false)
        }
      }

    handleSaveClick = () => {
        for(var i = 0; i < this.props.allinfo.length; i++) {
            var obj = this.props.allinfo[i];
            let json = JSON.stringify(obj)
            console.log(json)
        }
        this.handleCancelClick();
    }

    handleCancelClick = () => {
        this.props.setCancelButtonActive(true)
        this.props.setReset()
    }

    handleDisable = () => {
        if(this.props.active)
            return false
        else
            return true
    }

    render() {
        return (
            <Col lg="2" className="SaveCancel">
            <div>
                <Button onClick={this.handleSaveClick} className="save" color="primary" disabled={this.handleDisable()} size="lg">Save ></Button>
                <Button onClick={this.handleCancelClick} className="cancel" color="link" disabled={this.handleDisable()} size="lg">Cancel ></Button>
            </div>
            </Col>
        );
    }
}
```

Kuva 33. SaveCancel.js komponentti (Koskinen 2019.)

## 5.4 Komponenttien tyyli

Komponenttien tyylejä ei muokattu niinkään paljon vaan enimmäkseen värejä ja fontteja. Osittain siitä syystä, että React Bootstrap 4 komponentit olivat jo valmiiksi erittäin hyvin tyylitelty. Tyylistä otettiin mallia käyttöliittymän luonnoksesta. Rajojen lisääminen aiheutti pientä ongelmaa Fault komponentin kanssa, koska kyseinen komponentti oli ainoa missä sarakkeiden välillä ei ollut rajaviivaa. Tärkein tyyliin liittyvä asia oli responsiivisuus: käyttöliittymän haluttiin mahtuvan keskisuurille laitteille/näytöille eli laitteille, joiden näytön leveyden koko on minimissään 992 pikseliä. Tämä toteutettiin, mutta ei ilman ongelmia. Järjestelmässä, johon käyttöliittymä tehtiin, oli vielä kehitys vaiheessa, joten järjestelmään haluttiin päivittää Bootstrap 3 versiosta uuteen Bootstrap 4 versioon. Tämä päivitys vaikutti enemmän, kuin luultiin. Esimerkiksi tyylimäärittelyitä oli päällekkäin, joka vaikutti haastavasti käyttöliittymän responsiivisuuden muuttamiseen. Kuvassa 34 nähdään käyttöliittymän komponenttien ja tyylien lopullinen muoto.




Kuva 34. Renkaan tarkastusaseman uusi käyttöliittymä (Koskinen 2019.)

## 5.5 Viivakoodin luku

### 5.5.1 Implementointi

Toteutuksen alussa testattiin viivakoodinlukijan käsittelyä, jonka ansiosta toteutettiin ensimmäinen versio viivakoodinlukijan käsittelijästä. Tätä versiota lähdettiin parantelemaan ja kehittämään lisää. Viivakoodinlukijan käsittelijälle tehtiin oma luokka, KeyInputListener, jossa koko käsittely tapahtuu. KeyInputListener-luokka ottaa parametrina regexToListen, prefix, suffix ja funktion. RegexToListen oli tarkoituksena olla alfanumeerinen tarkastus, mutta todettiin että viivakoodinlukemisen jälkeen ei ole enää tarvetta tarkastelulle, vaan etu- ja jälkiliite riittävät tarkastuttamaan skannatun viivakoodin. Kyseinen parametri kuitenkin jätettiin koodin, jos esimerkiksi tulevaisuuden kehitysvaiheessa toimintaan tarvitaan muutosta. Parametrit prefix ja suffix ovat jo ensimmäisestä versiosta tuttuja: työssä käytetyssä viivakoodinlukijassa nämä olivat Controlb ja Controlc. Funktio parametrin tarkoituksena on viivakoodin löydyttyä ”laukaista” kyseinen funktio. Muuten toiminnallisuus ei kovin paljoa muuttunut ensimmäisestä versiosta. KeyInputListener-luokassa track funktion tarkoituksena on

laskea, että viivakoodi skannaukseen saa mennä vain tietty aika, jotta vältytään esimerkiksi käyttäjän mahdollisista vahinko etu- ja jälkiliitteiden painalluksilta. KeyInputListener-luokka luodaan aina sovelluksen Redux-sagassa, QControlSagas.js tiedostossa.

```
export default class KeyInputListener {
    constructor(options = {
        regexToListen: 'aa', triggeredFunc: defaultTriggeredFunc, prefix: 'Controlb', suffix: 'Controlc'
    }) {
        if (options.regexToListen && options.prefix && options.suffix) {
            this.regexToListen = new RegExp(options.regexToListen);
            this.prefix = options.prefix;
            this.suffix = options.suffix;
        }
        else {
            throw new Error('Please provide all options!');
        }
        this.triggeredFunc = options.triggeredFunc || defaultTriggeredFunc;
        this.trackingSince = null;
        this.isActive = false;
        this.trackingWindowMillis = 5000;
        this.keyBuffer = [];
    }
    handleTrackEvent = (event) => {
        this.track(event);
    }
    activate() {
        window.addEventListener('keydown', this.handleTrackEvent, true);
        this.isActive = true;
    }
    deactivate() {
        window.removeEventListener('keydown', this.handleTrackEvent, true);
        this.isActive = false;
    }
    move(arr, val) {
        var j = 0;
        for (var i = 0, l = arr.length; i < l; i++) {
          if (arr[i] !== val)
            arr[j++] = arr[i];
        }
        arr.length = j;
      }
    evaluateSequence() {
        let reg = new RegExp("(?<=" + this.prefix + ")(.*)(?=" + this.suffix + ")", "g")
        let regexBlur = new RegExp(/^Controlb/)
        this.move(this.keyBuffer, "Shift")
        if(regexBlur.test(this.keyBuffer.join('')))
            getStore().dispatch(qControlBlurActive(true))
        let result = this.keyBuffer.join('').match(reg)
        if(result != null) {
            let sequence = result.join('');
            this.trackingSince = null;
            this.keyBuffer = [];
            this.triggeredFunc(sequence);
        }
    }
    track(event) {
        if(this.isActive) {
            if(!this.trackingSince) {
                this.trackingSince =  new Date().getTime();
                this.keyBuffer.push(event.key);
                this.evaluateSequence();
            }
            else {
                const nextKeyTime = new Date().getTime();
                if(nextKeyTime < this.trackingSince + this.trackingWindowMillis) {
                    this.keyBuffer.push(event.key);
                    this.evaluateSequence();
                }
                else {
                    this.trackingSince =  new Date().getTime();
                    this.keyBuffer = [];
                    this.keyBuffer.push(event.key);
                    this.evaluateSequence();
                }
            }
        }
    }
}
```

Kuva 35. KeyInputListener.js luokka (Koskinen 2019.)

### 5.5.2 Toiminnot viivakoodinlukijalla

Tässä työssä yksi tärkeimmistä toiminnallisuuksista oli kyky käyttää käyttöliittymää melkein pelkästään viivakoodinlukijaa käyttäen. Kaikki muut toiminnot paitsi kommentin kirjoittaminen ja laatuvirheen poistaminen taulukosta piti pystyä tekemään viivakoodinlukijalla. Suurin osa tämän toiminnallisuuden toteutuksesta tehtiin Redux-sagassa. Apuna toteutuksessa käytettiin barcodes.json tiedostoa, johon laitettiin kaikkien haluttujen toimintojen viivakoodit ja muut tarvittavat tiedot (Kuva 36).


```
{"group": "10", "barcode": "CODE11.1", "code": 11.1}, {"group": "10", "barcode": "CODE11.2", "code": 11.2},
{"group": "10", "barcode": "CODE13.1", "code": 13.1}, {"group": "10", "barcode": "CODE15.1", "code": 15.1},
{"group": "10", "barcode": "CODE16.1", "code": 16.1}, {"group": "10", "barcode": "CODE17.1", "code": 17.1},
{"group": "10", "barcode": "CODE19.1", "code": 19.1},

{"group": "30", "barcode": "CODE31.1", "code": 31.1}, {"group": "30", "barcode": "CODE32.2", "code": 32.2},
{"group": "30", "barcode": "CODE33.1", "code": 33.1}, {"group": "30", "barcode": "CODE35.1", "code": 35.1},
{"group": "30", "barcode": "CODE35.2", "code": 35.2}, {"group": "30", "barcode": "CODE35.3", "code": 35.3},
{"group": "30", "barcode": "CODE36.1", "code": 36.1}, {"group": "30", "barcode": "CODE38.1", "code": 38.1},
{"group": "30", "barcode": "CODE38.2", "code": 38.2},

{"group": "40", "barcode": "CODE42.1", "code": 42.1}, {"group": "40", "barcode": "CODE42.2", "code": 42.2},
{"group": "40", "barcode": "CODE43.4", "code": 43.4}, {"group": "40", "barcode": "CODE44.1", "code": 44.1},
{"group": "40", "barcode": "CODE44.2", "code": 44.2}, {"group": "40", "barcode": "CODE45.1", "code": 45.1},
{"group": "40", "barcode": "CODE45.2", "code": 45.2}, {"group": "40", "barcode": "CODE45.3", "code": 45.3},
{"group": "40", "barcode": "CODE46.1", "code": 46.1}, {"group": "40", "barcode": "CODE47.1", "code": 47.1},

{"group": "50", "barcode": "CODE51.4", "code": 51.4}, {"group": "50", "barcode": "CODE52.1", "code": 52.1},
{"group": "50", "barcode": "CODE52.2", "code": 52.2}, {"group": "50", "barcode": "CODE52.3", "code": 52.3},
{"group": "50", "barcode": "CODE54.1", "code": 54.1}, {"group": "50", "barcode": "CODE52.1", "code": 52.1},

{"group": "60", "barcode": "CODE61.1", "code": 61.1}, {"group": "60", "barcode": "CODE61.2", "code": 61.2},
{"group": "60", "barcode": "CODE61.3", "code": 61.3}, {"group": "60", "barcode": "CODE62.1", "code": 62.1},
{"group": "60", "barcode": "CODE62.2", "code": 62.2}, {"group": "60", "barcode": "CODE65.2", "code": 65.2},
{"group": "60", "barcode": "CODE66.1", "code": 66.1}, {"group": "60", "barcode": "CODE66.3", "code": 66.3},
{"group": "60", "barcode": "CODE66.4", "code": 66.4}, {"group": "60", "barcode": "CODE69.1", "code": 69.1},

{"group": "70", "barcode": "CODE71.1", "code": 71.1}, {"group": "70", "barcode": "CODE72.1", "code": 72.1},
{"group": "70", "barcode": "CODE74.1", "code": 74.1}, {"group": "70", "barcode": "CODE74.2", "code": 74.2},
{"group": "70", "barcode": "CODE74.4", "code": 74.4}, {"group": "70", "barcode": "CODE75.1", "code": 75.1},
{"group": "70", "barcode": "CODE75.2", "code": 75.2}, {"group": "70", "barcode": "CODE79.1", "code": 79.1},

{"group": "80", "barcode": "CODE81.1", "code": 81.1}, {"group": "80", "barcode": "CODE82.1", "code": 82.1},
{"group": "80", "barcode": "CODE82.3", "code": 82.3}, {"group": "80", "barcode": "CODE83.1", "code": 83.1},
{"group": "80", "barcode": "CODE84.1", "code": 84.1},

{"group": "90", "barcode": "CODE91.1", "code": 91.1}, {"group": "90", "barcode": "CODE91.2", "code": 91.2},
{"group": "90", "barcode": "CODE91.2", "code": 91.2}, {"group": "90", "barcode": "CODE91.3", "code": 91.3},
{"group": "90", "barcode": "CODE93.1", "code": 93.1}, {"group": "90", "barcode": "CODE94.1", "code": 94.1},
{"group": "90", "barcode": "CODE95.1", "code": 95.1}, {"group": "90", "barcode": "CODE95.2", "code": 95.2},
{"group": "90", "barcode": "CODE98.1", "code": 98.1},

{"group": "200", "barcode": "CODE204.1", "code": 204.1}, {"group": "200", "barcode": "CODE216.1", "code": 216.1},
{"group": "200", "barcode": "CODE225.1", "code": 225.1},
```


Kuva 36. Kuvakaappaus barcodes.json tiedostosta (Koskinen 2019.)

Tässä työssä Redux-sagaa käytettiin viivakoodin hallintaan. Käyttöliittymän Redux-saga sijaitsee QControlSagas.js tiedostossa (Kuva 37). KeyInputListener-luokka luodaan Redux-sagassa ja jokainen luettu viivakoodi menee ensin Redux-sagan lävitse. Funktio handleBarcodeRead tarkoitus on käsitellä kaikki hyväksytyt viivakoodit, eli ne viivakoodit, joista löytyy oikeat etu- ja jälkiliitteet. Kun handleBarcodeRead funk-

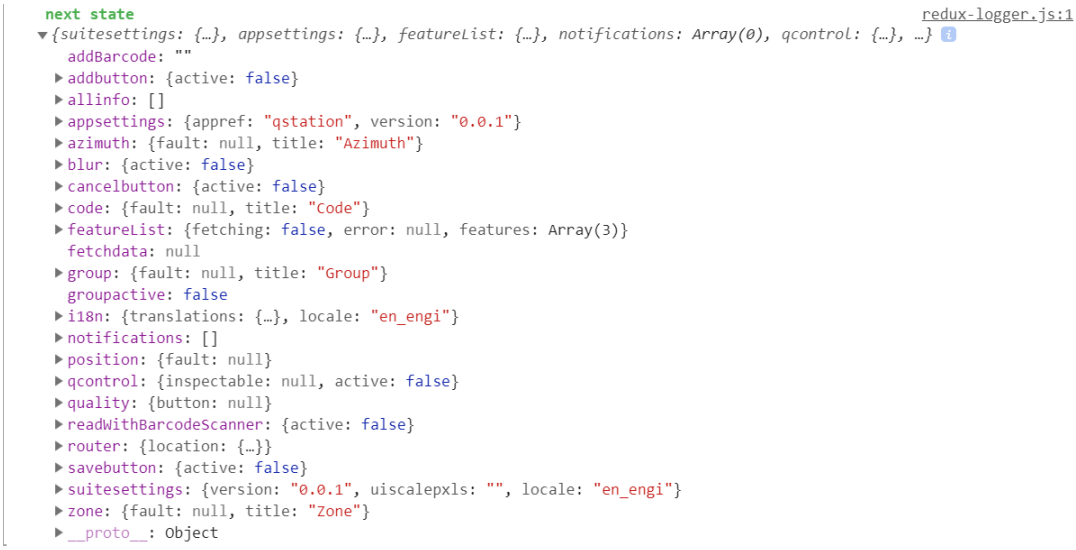
tio havaitsee hyväksytyn viivakoodin se lähettää actionin reducerille. Lähetettävä actioni riippuu funktioon tulevasta viivakoodista. Esimerkiksi jos, funktioon tulee ”ADD” merkkijono/viivakoodi Redux-saga lähettää actionin reducerille, mikä päivittää CommentAreassa olevan Add-napin aktiiviseksi. CommentArea komponentti päivittyy ja huomaa että Add-napin tila on aktiivinen, joten tehdään toiminto, mikä tässä esimerkissä olisi Add-napin painallus eli laatuvirheiden tallentaminen allinfo tilaan.

```
let keyInputListener = null;
// eslint-disable-next-line
function* initKeyEventListener() {
    keyInputListener = new KeyInputListener({
        regexToListen: /^[a-z0-9]+$/i,
        prefix: 'Controlb',
        suffix: 'Controlc',
        triggeredFunc: (sequence) =>{
            getStore().dispatch(qControlBarcodeRead(sequence))
        }});
    keyInputListener.activate();
}

function* handleBarcodeRead(action) {
    const barcodeFound = barcodes.find(button => button.barcode === action.barcode)
    let getBarcodeActive = (state) => state.qcontrol.active
    let isActive = yield select(getBarcodeActive)

    if(barcodeFound) {
        if(barcodeFound.barcode.includes("PS") && isActive) {
            yield put(qControlFaultPosition(barcodeFound.button))
        }
        else if(barcodeFound.barcode.includes("CL") && isActive) {
            yield put(qControlQualityButtons(barcodeFound.button))
        }
        else if(barcodeFound.barcode.includes("ADD") && isActive) {
            yield put(qControlAddButtonActive(barcodeFound.active))
        }
        else if(barcodeFound.barcode.includes("SAVE") && isActive) {
            yield put(qControlSaveButtonActive(barcodeFound.active))
        }
        else if(barcodeFound.barcode.includes("CODE") && isActive) {
            yield put(qControlFaultGroup(barcodeFound.group, barcodeFound.group))
            yield put(qControlGroupActive(true))
            yield put(qControlFaultCode(barcodeFound.code, barcodeFound.code))
        }
        else if(barcodeFound.barcode.includes("ZONE") && isActive) {
            yield put(qControlFaultZone(barcodeFound.zone, barcodeFound.zone))
        }
        else if(barcodeFound.barcode.includes("AZIM") && isActive) {
            yield put(qControlFaultAzimuth(barcodeFound.azimuth, barcodeFound.azimuth))
        }
    }
    else {
        yield put(qControlSetBarcodeToInspect(action.barcode, true))
        yield put(qControlFetchData(action.barcode))
        yield put(qControlReadWithBarcodeScanner(true))
    }
}
function* barcodeReadWatcher() {
    yield takeLatest(Q_CONTROL_BARCODE_READ, handleBarcodeRead)
}

export default function* AppSagas() {
    yield fork(initKeyEventListener);
    yield fork(barcodeReadWatcher);
}

```

Kuva 37. QControlSagas.js tiedoston koodi

## 5.6 Tilan hallinta

Tässä työssä oli tarkoituksena käyttää Reduxin ominaisuuksia mahdollisimman hyödyllisesti. Kaikki mahdollinen pyrittiin laittamaan Redux storeen. Muutamia tiloja jäi storen ulkopuolelle komponenttien omiin tiloihin, mutta kaikki tärkeimmät arvot ja datat saatiin laitettua storeen. Kuvassa 38 näkyy käyttöliittymän kaikki storessa olevat tilat. Google Chromen laajennussovellusta, React Developers Toolsia, käytettiin paljon apuna tilojen testaamisessa. Tämän laajennuksen avulla, Redux store nähtiin koko ajan konsolissa luettavassa muodossa.




Kuva 38. Redux store (Koskinen 2019.)

# 6 YHTEENVETO

React-kirjasto, Redux ja Redux-saga eivät olleet itselleni kovinkaan tuttuja entuudestaan. Ainoa JavaScript-kirjasto, johon olin törmännyt verkkosivuja tehdessä, oli jQuery. Tästä syystä, tämän työn tekeminen vaati paljon itseopiskelua ja omaaloitteisuutta. Pohjana itselläni oli kuitenkin paljon erilaisia ohjelmointikieliä, joten vanhan pohjalta oli helppo luoda uutta.

Opinnäytetyötä valittaessa, Cimcorp tarjosi erilaisia vaihtoehtoja. Ajattelin, että web-toteutuksena toteutettava käyttöliittymä, johon kuului viivakoodinlukijan käsittely, olisi itselleni tarpeeksi haastava ja mielenkiintoinen. Mielenkiintoa lisäsi myös se, että kyseinen käyttöliittymä tulisi oikeasti käyttöön jossain vaiheessa. Haasteellista tässä työssä oli esimerkiksi viivakoodinlukijan käsittely ja Redux storen tilojen hallinta ja suunnittelu. Loppujen lopuksi kaikkiin haastavampiinkin asioihin löydettiin aina ratkaisu, joten mikään ei ollut ylitsepääsemätöntä.

Tässä työssä pidin myös tärkeänä versionhallintaa, vaikka kehittäjiä työssä oli vain yksi. En ollut entuudestaan kovin paljon versionhallintaa käyttänyt, joten tämä sopi itselleni vallan mainiosti. Yleensä ohjelmistosuunnittelija tai kehittäjä ei tee työtä yksin, ja niinpä esimerkiksi Gitin ja Bitbucketin käytön osaaminen on mielestäni erittäin tärkeää. Tämän olen huomannut nyt Cimcorpin projektipuolen Software-osastolla työskennellessäni.

Tämä työ oli kaiken kaikkiaan opettavainen, motivoiva ja mielenkiintoinen kokemus. Työ kasvatti mielenkiintoa erilaisiin web-toteutuksiin ja sovelluksiin. Erilaisten tekniikoiden ja kirjastojen itseopiskelu on jatkunut työn valmistumisen jälkeenkin. Olen tyytyväinen työn lopputulokseen, käytettyjen tekniikoiden oppimiseen ja omaksumiseen.

# LÄHTEET


Atto, E. 2018. A Beginner's Guide to React with ES6. Viitattu 22.3.2019. https://medium.com/the-andela-way/a-beginners-guide-to-react-with-es6-a2ed0b5c977e

BabelJS. 2018. What is Babel? Viitattu 25.3.2019. https://babeljs.io/docs/en/

Banks, A. & Porcello, E. 2017. Learning React. Viitattu 22.3.2019.

Beattie, S. 2018. What is ECMAScript 6? Viitattu 22.3.2019. https://www.quora.com/What-is-ECMAScript-6

Buddy. 2016. 5 types of Git workflow that will help you deliver better code. https://buddy.works/blog/5-types-of-git-workflows

Eisenman, B. 2015. Chapter 2. Working with React Native. Viitattu 24.3.2019. https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html

Elder, R. 2016. What is Git and Why Is It Useful? Viitattu 24.3.2019. http://blog.robertelder.org/what-is-git/

Ighodaro, N. 2018. Why use Redux? Reasons with clear examples. Viitattu 24.3.2019. https://blog.logrocket.com/why-use-redux-reasons-with-clear-examples-d21bffd5835

JavaScript.info. 2019. An Introduction to JavaScript. Viitattu 20.3.2019. https://javascript.info/intro

Kagga, J. 2018. Understanding React Components. Viitattu 27.3.2019. https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb

Karwal, A. 2018. React Application Data Flow with Redux. Viitattu 27.3.2019. https://medium.com/@ajaykarwal/react-application-data-flow-with-redux-1ad51aa4ac45

Mozilla. 2019. How CSS works. Viitattu 28.3.2019. https://developer.mozilla.org/en-US/docs/Learn/CSS/Introduction_to_CSS/How_CSS_works

Rascia, T. 2015. What is Bootstrap and How Do I Use It? Viitattu 28.3.2019. https://www.taniarascia.com/what-is-bootstrap-and-how-do-i-use-it/

Ravichandran, A. 2018. React Virtual DOM Explained in Simple English. Viitattu 28.3.2019. https://programmingwithmosh.com/react/react-virtual-dom-explained/

ReactJS. 2019. JSX In Depth. Viitattu 29.3.2019. https://reactjs.org/docs/jsx-in-depth.html

Reactstrap. 2019. About the Project. Viitattu 30.3.2019. https://reactstrap.github.io/

Redux-saga. 2019. redux-saga. Viitattu 10.4.2019. https://redux-saga.js.org/

Rich, R. 2016. Document Object Model Nasrullah. DOM When a page is loaded,browser creates a Document Object Model of the Page. Viitattu 20.3.2019. https://slideplayer.com/slide/9631242/

Sahin, K. 2017. 7 Reasons why you should use React. Viitattu 24.3.2019. https://stories.jotform.com/7-reasons-why-you-should-use-react-ad420c634247

Sharma, S. 2018. Declarative Programming & React. Viitattu 26.3.2019. https://dev.to/itsjzt/declarative-programming--react-3bh2

SixBit. 2019. Setting Up Your Barcode Scanner. Viitattu 3.4.2019. https://www.sixbitsoftware.com/EN/Barcode_Scanners.htm

Sridhar, A. 2018. An introduction to Git: what it is, and how to use it. Viitattu 14.4.2019. https://medium.freecodecamp.org/what-is-git-and-how-to-use-it-c341b049ae61

Vishwakarma, V. 2016. What is Bitbucket? Viitattu 17.4.2019. https://www.quora.com/What-is-Bitbucket

Weck, S. 2017. Developing modern offline apps with ReactJS, Redux and Electron – Part 3 – ReactJS + Redux. Viitattu 27.3.2019. https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-basics/